TURING 图灵程序设计丛书



The Art of Debugging with GDB, DDD, and Eclipse

软件调试的艺术

[美] Norman Matloff
Peter Jay Salzman 著
张云 译

GDB



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

 图灵程序设计丛书

软件调试的艺术

The Art of Debugging with GDB, DDD, and Eclipse

[美] Norman Matloff
Peter Jay Salzman 著

张 云 译

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

软件调试的艺术 /（美）马特洛夫（Matloff, N.），（美）萨尔兹曼（Salzman, P. J.）著；张云译．—北京：人民邮电出版社，2009.11
（图灵程序设计丛书）
书名原文：The Art of Debugging with GDB, DDD, and Eclipse
ISBN 978-7-115-21396-9

Ⅰ. 软… Ⅱ. ①马…②萨…③张… Ⅲ. 软件－调试 Ⅳ. TP311.5

中国版本图书馆CIP数据核字（2009）第162737号

内 容 提 要

调试对于软件的成败至关重要，正确使用恰当的调试工具可以提高发现和改正错误的效率。本书详细介绍了3种调试器，GDB用于逐行跟踪程序、设置断点、检查变量以及查看特定时间程序的执行情况，DDD是流行的GDB的GUI前端，而Eclipse提供完整的集成开发环境。书中不但配合实例讨论了如何管理内存、理解转储内存、跟踪程序找出错误等内容，更涵盖了其他同类书忽略的主题，例如线程、客户/服务器、GUI和并行程序，以及如何躲开常见的调试陷阱。

本书适合各层次软件开发人员、管理人员和测试人员阅读。

图灵程序设计丛书

软件调试的艺术

◆ 著 [美] Norman Matloff Peter Jay Salzman
译 张 云
责任编辑 傅志红

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：14.25
字数：337千字 2009年11月第1版
印数：3 001－5 000册 2010年1月北京第2次印刷

著作权合同登记号 图字：01-2008-5834号

ISBN 978-7-115-21396-9

定价：39.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

前　言

"嘿，还真不错呢！"我们的学生Andrew首次认真使用调试工具后惊叹道。Andrew三年前在大一上编程课时就学过了调试工具，不过那时他只是将调试工具当做应付期末考试的内容来打发的。现在他已经大四了，教授强烈要求他停止采用输出语句进行调试的方法，改为使用正式调试工具。让Andrew喜出望外的是，他发现使用恰当的工具可以大大缩短调试时间。

如今，在学生及已参加工作的程序员中，还有不少"Andrew"，但愿本书能帮助"Andrew们"发现调试工具的好处。但是，我们更希望本书能帮助那些已经使用了调试工具，但还无法确定在什么情况下该做什么事的程序员做出适当的判断。本书也适用于打算进一步学习调试工具及其幕后原理的人。

本书的编辑曾说过，很多调试知识以前都只是在圈子里口口相传，没有正式编著成书。本书将改变这一状况。本书回答了下列问题。

- 如何调试线程代码？
- 为什么有时候断点最终出现的位置与原来设置的位置略有偏差？
- GDB的`until`命令为什么有时会跳到意想不到的地方？
- 如何巧妙使用DDD和Eclipse？
- 在当今普遍使用GUI的时代，像GDB这样的基于文本的应用程序还有价值吗？
- 为什么当错误代码超出数组边界时不发生段错误？
- 为什么要将我们的一个示例数据结构命名为`nsp`？（不好意思，这只是与我们的出版商No Starch Press私下开的一个玩笑。）

本书既没有美其名曰"用户手册"，也不是关于调试过程认知理论的抽象论文。本书有点介于这两者之间。一方面，确实提供了如何操作GDB、DDD和Eclipse中特定命令的信息；另一方面，讲解并频繁使用了关于调试过程的一些通用原则。

我们之所以选择GDB、DDD和Eclipse，是因为它们在Linux/开源社区中比较流行。本书的示例有点偏向于GDB，不仅仅因为GDB基于文本的性质使得显示在一个页面中更紧凑，而且正如上面的问题所暗示的，我们发现基于文本的命令在调试过程中仍然起着举足轻重的作用。

Eclipse的用途越来越广泛，其远不止仅作为我们这里的调试角色，还提供了各种有吸引力的调试工具。另一方面，DDD占用空间小，而且包括了Eclipse中不具备的某些强大功能。

第1章是概览。很多经验丰富的程序员可能想跳过这一章，但是我们强烈建议大家通读一遍，因为这一章列出了我们针对调试过程推荐的一些简单却有用的通用准则。

第2章着重介绍了调试必不可少的内容——断点，讨论了关于断点的所有细节，包括设置、删除和禁用断点，从一个断点移到另一个断点，查看关于断点的详细信息，等等。

到达断点后会出现什么情况呢？第3章回答了这一问题。我们这里采用的示例涉及遍历树的代码，重点是介绍当到达断点时如何方便地显示树中节点的内容。这里GDB相当出色，提供了一些非常灵活的功能，有助于在每次程序暂停时有效地显示感兴趣的信息。这一章还提供了一个特别优秀的用图形显示树和其他链接数据结构的DDD功能。

第4章包括了由于段错误而产生的致命运行时错误。我们首先介绍了崩溃时在底层发生了什么事，包括程序的内存分配以及硬件与操作系统的协同作用。对系统知识比较了解的读者可能会跳过这一章，但是我们相信，其他很多读者会得益于这一章介绍的基础知识。然后介绍了核心文件，包括如何创建核心文件，如何使用它们来完成“事后反思”。最后介绍了关于调试会话的一个扩展示例，其中有几个程序错误产生了段错误。

第5章不但介绍并行编程，而且包括网络代码。客户/服务器网络编程可算作并行处理，甚至我们的工具也是并行使用的。比如，在两个窗口中使用GDB，一个窗口用于客户机，另一个窗口用于服务器。由于网络代码涉及系统调用，因此我们用C/C++的`errno`变量和Linux的`strace`命令补充我们的调试工具。5.2节涉及线程编程。这里同样首先概述基本内容：分时、进程与线程、竞争条件等。这一章介绍了在GDB、DDD和Eclipse中使用线程的技术细节，并再次讨论了一些要记住的通用原则，比如发生线程上下文切换时的时间选择随机性。5.4节介绍了用流行的MPI和OpenMP程序包进行并行编程，并举了一个OpenMP的扩展示例。

第6章包括其他一些重要主题。如果代码不能编译，那么调试工具将无能为力，因此这一章讨论了处理这种问题的一些方法。然后处理由于缺少必要的库造成的连接失败问题，我们再一次觉得提供一些“理论”是有用的，比如库的类型以及如何将库与主要代码连接。如何调试GUI程序呢？为了简便起见，我们这里坚持采用“半GUI”设置——curses编程，并显示如何让GDB、DDD和Eclipse与curses窗口中的事件交互。

正如前面提到的，可以使用辅助工具使调试过程得到很大的增强，第7章介绍了部分辅助工具，还介绍了`errno`和`strace`、`lint`的一些内容，以及对于有效使用文本编辑器的一些提示。

全书主要介绍的是C/C++编程的调试，第8章则谈到了其他语言，包括Java、Python、Perl和汇编语言。

如果漏掉了读者喜欢的调试主题，我们感到抱歉，书中包括了我们自己编程时发现有用的全部内容。

感谢No Starch Press的诸位工作人员在这个项目上花了很长时间来协助我们。尤其感谢公司的创始人及总编辑Bill Pollock。他一开始就对我们这个“非常规”项目抱有信心，并宽容了我们的多次延误。

Daniel Jacobowitz对本书的初稿做了认真的审查，并提出了许多宝贵建议。还有Neil Ching，虽然他表面上是做文字编辑的，实际上却是拥有计算机学士学位的高材生。他在我们的技术讨论中提出了一些重要看法。本书质量的提高很大程度上得益于从Daniel和Neil处得到的反馈。当然，照例要做一下免责声明：如果还有错误，那一定是我们自己造成的。

Norm的致谢。感谢我的妻子Gamis和女儿Laura，这两个可爱的女子让我觉得非常幸福。尽管这本书她们一个字也没看过，但是她们解决问题的方法、幽默的性格和对生活的热爱深深地影响了我。还要感谢这么多年来我教过的学生，他们教会我的与我教会他们的一样多，是他们让我觉得我选对了职业。我一直致力于“有所作为”，但愿这本书能算作我的一个小小作为。

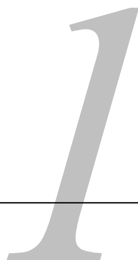
Pete的致谢。我还要感谢Nicole Carlson、Mark Kim和Rhonda Salzma，花了不少时间来通读本书的各章，提出了更正与建议，感谢你们阅读本书。还要感谢Davis的Linux用户组上这么多年来回答我问题的人们，认识你们使我更聪明。感谢Evelyn提升了我生活的方方面面。特别值得一提的是小猫咪Geordi，它总是趴在我的稿纸上，以免稿子被吹散，还时常为我温暖座椅，并值守书房。我每天都在想念你们。小家伙，睡吧。妈妈，看儿子棒吗？

Norm Matloff与Pete Salzman
于2008年6月9日

目　录

第 1 章

预备知识



有些读者，特别是专业人士，可能想要跳过本章内容。然而，我们建议每个人都应该至少简单浏览一下。许多专业人士会从中发现一些对于他们来说是全新的内容。无论如何，所有读者都应该熟悉这些内容，这一点很重要，因为其后的各章都将用到这些内容。当然，初学者更应该仔细地阅读本章。

本章前几节将概述调试过程并介绍调试工具的作用，然后在1.7节给出一个扩展性示例。

1.1 本书使用的调试工具

本书中，我们将阐明调试的基本原则，并且在如下调试工具的环境中说明这些基本原则。

- GDB

Unix程序员最常用的调试工具是GDB，这是由Richard Stallman（开源软件运动的领路人）开发的GNU项目调试器（GNU Project Debugger），该工具在Linux开发中扮演了关键的角色。

大多数Linux系统应该预先安装了GDB。如果没有预先安装该工具，则必须下载GCC编译器程序包。

- DDD

随着GUI（图形用户界面）越来越流行，大量在Unix环境下运行的基于GUI的调试器被开发出来。其中的大多数工具都是GDB的GUI前端：用户通过GUI发出命令，GUI将这些命令传递给GDB。DDD（Data Display Debugger，数据显示调试器）就是其中的一种工具。

如果你的系统还没有安装DDD，则可以下载该工具。例如，在Fedora Linux系统上，命令

```
yum install ddd
```

将自动处理整个安装过程。在Ubuntu Linux上，可以使用命令`apt-get`。

- Eclipse

有些读者可能使用过IDE（集成开发环境）。IDE也是一种调试工具，它在一个程序包中集成了编辑器、生成工具、调试器和其他开发辅助程序。本书中的示例IDE是非常流行的Eclipse系统。与DDD一样，Eclipse在GDB或其他一些调试器的基础上运行。

可以通过如上所述的`yum`或`apt-get`命令安装Eclipse，也可以直接下载相应的*.zip*文件，在适

当的目录（例如*/user/local*）中解压缩该文件。

本书使用3.3版本的Eclipse系统。

1.2　编程语言

本书主要面向C/C++编程，并且将在该环境中完成大多数示例。不过，第8章也会讨论其他语言。

1.3　调试的原则

虽然调试是一门艺术而非科学，但是仍然有一些明确的原则来指导调试的实践。本节将讨论其中一些原则。

至少其中的一个规则，即确认的基本原则（Fundamental Principle of Confirmation）实际上是相当有效的原则。

1.3.1　调试的本质：确认原则

下面的规则说明了调试的本质。

- 确认的基本原则

修正充满错误的程序，就是逐个去确认你认为正确的代码确实是正确的。当你发现其中某个假设不成立时，就表示已经找到了关于程序错误所在位置的线索（甚至是错误本身）。

换一种表达方式来说：惊讶是好事！

当你认为关于程序的某件事情是正确的，但却不能确认它，你就会感到惊讶。但这种惊讶是好事，因为这种发现会引导你找到程序错误所在的位置。

1.3.2　调试工具对于确认原则的价值所在

传统的调试技术只是向程序中添加跟踪代码以在程序执行时输出变量的值，例如使用`printf()`或`cout`语句输出变量的值。你可能会问："这样操作还不够吗？为什么要使用GDB、DDD或Eclipse这样的调试工具？"

首先，这种方法要求有策略地持续添加跟踪代码，重新编译程序，运行程序并分析跟踪代码的输出，在修正程序错误之后删除跟踪代码，并且针对发现的每个新的程序错误重复上述这些步骤。这种工作过程非常耗时，并且容易令人疲劳。最为重要的是，这些操作将你的注意力从实际任务转移开，使你无法集中精力通过严密推理去查找程序错误。

相反，通过使用DDD和Eclipse这样的图形化调试工具，只需要将鼠标指针移动到代码显示中的变量实例上方就可以检查该变量的值，并且此时会显示该变量的当前值。为什么还要在整夜的调试中使用`printf()`语句来输出变量的值，毫无必要地让自己更加疲劳，等待更长的时间呢？不如放松心情，使用调试工具可以减少所花费的时间，减轻你需要忍受的厌烦感。

使用调试工具不仅仅能够查看变量。在许多情况下，调试器会指出程序错误所在的大概位置。例如，程序由于段错误（即内存访问错误）而崩溃。在本章后面的样本调试会话中可以看到，

GDB/DDD/Eclipse可以立刻指出段错误所在的位置，这一般就是（或接近于）程序错误所在的位置。

类似地，调试器要求用户设置监视点（watchpoint），通过监视点可以了解在程序运行期间某个变量的值到达可疑值范围的具体位置，而不用调试器，只查看调用`printf()`获得的输出很难推断出这种信息。

1.3.3　其他调试原则

● 从简单工作开始调试

在调试过程的开始阶段，应该从容易、简单的情况开始运行程序。这样做也许无法揭示所有的程序错误，但是很有可能发现其中的部分错误。例如，如果代码由大型循环组成，则最容易发现的程序错误是在第一次或第二次迭代期间引发的程序错误。

● 使用自顶向下的方法

你可能已经了解如何使用自顶向下或模块化的方法来编写代码：主程序不应该过长，它应该去调用那些执行大量实际工作的函数。如果某个执行实际工作的函数较长，则应该考虑将该函数依次分解为多个较小的模块。

除了应该以自顶向下的方式编写代码之外，也应该以这种方式调试代码。

例如，假设程序使用函数`f()`。在使用调试工具深入代码并且遇到对`f()`的调用时，调试器可让你选择执行过程中下一次暂停的位置——是在调用函数的第一行还是在函数调用后的下一条语句。在许多情况下，先选后一种比较好：执行调用，然后检查与调用有关的变量的值，从而了解该函数是否正确运行。如果该函数正确运行，就可以避免单步调试函数中代码这样既浪费时间又不必要的工作，因为这些代码并没有出错。

● 使用调试工具确定段错误的位置

当发生段错误时，执行的第一步操作应该是在调试器中运行程序并重新产生段错误。调试器将指出发生这种错误的代码行。然后，可以通过调用调试器的反向跟踪（backtrace）功能获得其他有用信息，该功能会显示出错前所有函数的调用序列。

在某些情况下，可能很难重新产生段错误，但是如果有核心文件（core file），则仍然可以执行反向跟踪以确定产生段错误的情况，第4章将讨论这个问题。

● 通过发出中断确定无限循环的位置

如果怀疑程序中存在无限循环，则进入调试器并再次运行程序，让该程序执行足够长的时间以进入循环。然后，使用调试器的中断命令挂起该程序，并且执行反向跟踪，了解已到达循环主体的哪个位置以及程序是如何到达该位置的。（该程序还没有被取消执行，可以根据需要恢复执行。）

● 使用二分搜索

你可能已经在有序列表的环境中看到过二分搜索。例如，假设你有一个数组`x[]`包含按升序排列的500个数字，并且希望确定在应何处插入新的数字`y`。首先将`y`与元素`x[250]`进行比较，如果结果是`y`小于该元素，接下来就将`y`与元素`x[125]`进行比较；但是如果`y`大于`x[250]`，则下一次就将`y`与`x[375]`进行比较。在后一种情况中，如果`y`小于`x[375]`，则将其与`x[312]`进行比较，`x[312]`是`x[250]`和`x[375]`之间的中间元素，以此类推。在每次迭代过程中保持将搜索范围减半，从而快

速查找到插入点。

在调试过程中也可以应用二分搜索的原理。假设你知道在最初的1 000次循环迭代期间，存储在某个变量中的值在某个时刻出现问题。那么一种方法是使用监视点跟踪该值第一次出现问题时所在的迭代过程，1.5.3节将讨论这种高级技术。另一种方法则是使用二分搜索，此时在时间而不是空间上进行二分。首先检查该变量在第500次迭代后的值，如果此时其值仍然良好，则下一次检查位于第750次迭代后的值，以此类推。

再举另一个例子，假设程序中的某个源文件根本无法编译。由语法错误产生的编译器消息中，提及的代码行有时与实际的错误位置相去甚远，因此可能很难确定该位置。在这种情况下，二分搜索就可以派上用场：删除（或注释掉）编译单元中的一半代码，重新编译剩余的代码并查看错误消息是否仍然存在。如果错误消息仍然存在，则错误就在这一半代码中；如果错误消息不再出现，则错误就位于删除的那一半代码中。一旦确定了包含程序错误的一半代码，就可以进一步将程序错误限制在这半部分代码的一半，继续执行相同的操作直到确定问题所在位置。当然，在开始该过程之前应该建立原始代码的副本，或者更好的方法是使用文本编辑器的撤销功能。参见第7章，了解在编程时适当地使用编辑器的技巧。

1.4 对比基于文本的调试工具与基于 GUI 的调试工具，两者之间的折中方案

本书讨论的GUI（DDD和Eclipse）充当用于C和C++的GDB以及其他调试器的前端。虽然GUI具有显眼的外观并且使用起来也比基于文本的GDB更为方便，但是本书主张的观点是基于文本的调试器和基于GUI的调试器（包括IDE）在不同环境下各有所长。

1.4.1 简要比较界面

为了快速了解基于文本的调试工具和基于GUI的调试工具之间的区别，来看看将它们用于本章示例的情况。该示例中的程序是*insert_sort*，通过源文件*ins.c*编译该程序，其完成的是插入排序。

1. GDB：纯文本

为了使用GDB启动针对这个程序的调试会话，可以在Unix命令行中输入如下命令：

```
$ gdb insert_sort
```

在此之后，GDB会通过显示如下提示符来邀请你提交命令：

```
(gdb)
```

2. DDD：GUI调试工具

使用DDD时，通过在Unix命令行中输入如下命令来开始调试会话：

```
$ ddd insert_sort
```

此时将打开DDD窗口，在此之后就可以通过GUI提交命令。

DDD窗口的一般外观如图1-1所示。可以看到，DDD窗口在各个子窗口中安排信息。

- 源文本窗口显示源代码。DDD从`main()`函数处开始显示，但是可以通过使用窗口右侧的滚动条移动到源文件的其他部分。
- 菜单栏提供各种菜单类别，包括File（文件）、Edit（编辑）和View（视图）。
- 命令工具列出最常见的DDD命令（例如Run、Interrupt、Step和Next），从而可以快速访问这些命令。
- 控制台：回顾一下，DDD只是GDB（和其他调试器）的GUI前端。DDD将鼠标选中的命令转换为对应的GDB命令。这些命令和它们的输出显示在控制台窗口中。此外，可以直接通过控制台窗口提交命令给GDB，这是非常方便的功能，因为并不是所有的GDB命令都有对应的DDD命令。
- 数据窗口显示已经请求连续显示的变量的值。在执行这样的请求之前，这个子窗口不会出现，因此它没有出现在图1-1中。

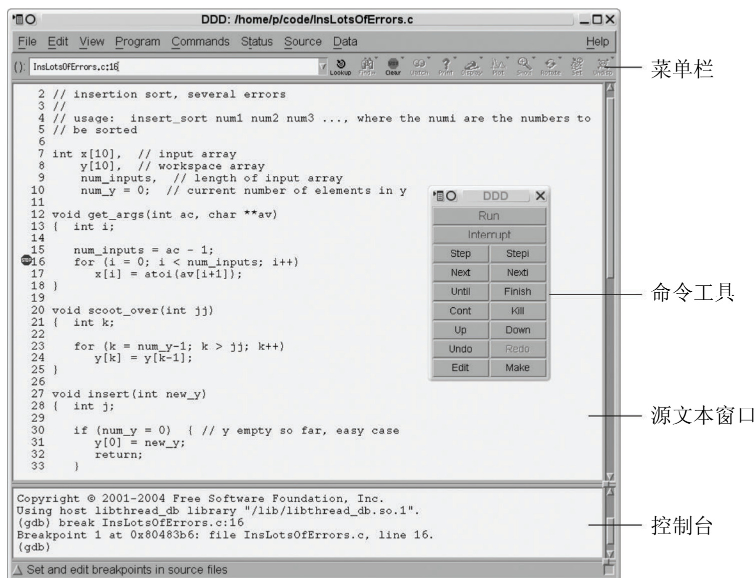


图1-1 DDD布局

下面简单演示在每种类型的用户界面下如何将调试命令提交给调试器。在调试*insert_sort*程序时，你可能希望暂停该程序的执行，从而在函数`get_args()`的第16行（假设）处设置断点（在1.7节中将看到*insert_sort*的完整源代码）。为了在GDB中设置断点，可以在GDB提示符中输入如下命令：

```
(gdb) break 16
```

完整的命令名是`break`，但是GDB允许在不产生歧义的情况下使用缩写，并且大多数GDB用

户会在此处输入b 16。考虑到GDB的初学者不易理解这些缩写，我们将先使用完整的命令名，在本书的后面，当用户已经较为熟悉这些命令时我们再使用缩写。

使用DDD，将查看源文本窗口，单击第16行的开始位置，然后单击DDD屏幕顶端的Break（中断）图标。也可以在第16行的开始位置右击，然后选择Set Breakpoint（设置断点）。另一种方法是在代码行开始位置左侧的任意位置双击该代码行。无论采用何种方法，DDD都会通过在该行中显示小的停止符号来确认选择，如图1-2所示。通过这种方式，可以快速浏览断点。

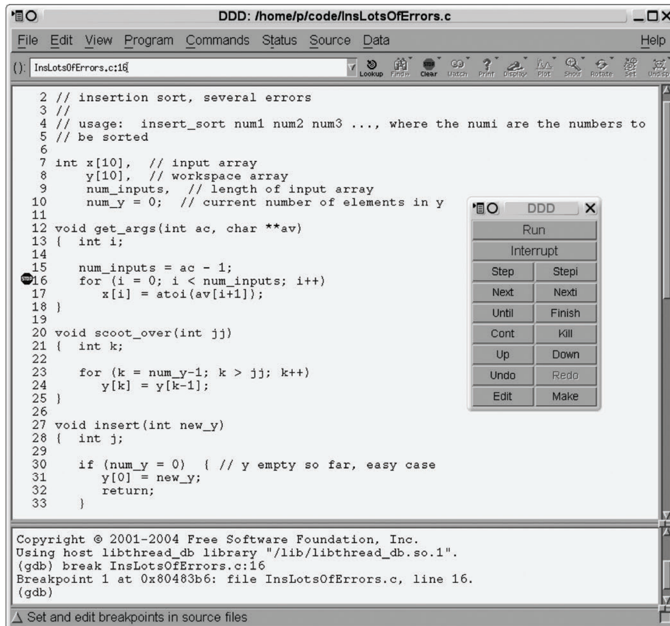


图1-2　断点设置

3. Eclipse：不只是GUI调试器

图1-3显示了Eclipse中的常规环境，按照Eclipse术语，我们当前正处于调试透视图（Debug Perspective）中。Eclipse是开发许多不同类型软件的常规框架。每种编程语言在Eclipse中都有自己的插件GUI，即透视图。实际上，对于同一种语言可能有多个竞争的透视图。在本书的Eclipse操作中，我们将使用用于C/C++开发的C/C++透视图、用于编写Python程序的Pydev透视图等。也有用于执行实际调试工作的调试透视图（带有一些语言特有的功能），图1-3就显示了该透视图。

C/C++透视图是CDT插件的一部分。类似于DDD的情况，CDT在后台调用GDB。

透视图的相关组件大体上类似于上面所描述的DDD的窗口组件。透视图被分解为多个称为视图的选项卡式窗口。位于透视图左侧的是源文件*ins.c*的视图，也有用于检查变量值的变量视图（这时该视图中还没有任何变量值），此外还有控制台视图，其功能非常类似于DDD中的同名子窗口，另外还有其他的视图。

可以按照与在DDD中相同的操作来可视化地设置断点。例如，在图1-4中，源文件窗口中的

代码行

```
for (i = 0; i < num_inputs; i++)
```

在其左侧页边空白处有一个蓝色符号，表示此处有一个断点。

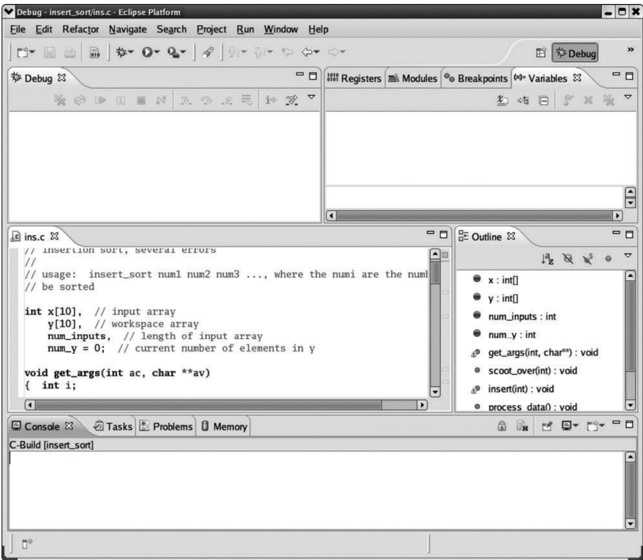


图1-3 Eclipse环境

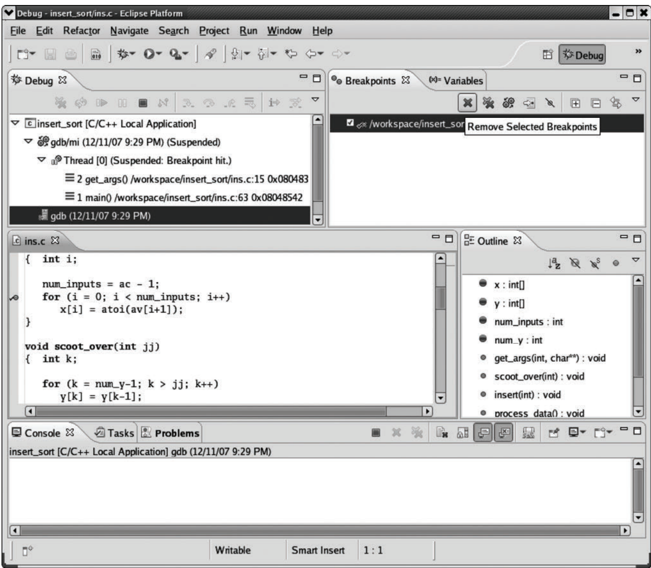


图1-4 在Eclipse中删除断点

4. Eclipse与DDD的对比

Eclipse也有一些DDD所没有的辅助功能。例如，Eclipse环境的右侧是Outline视图，该视图中列出了变量、函数等信息。如果单击函数`scoot_over()`对应的条目，则源文件视图中的光标将会移动到该函数上。此外，如果临时从调试透视图移动回正在其中对该项目执行编辑和编译的C/C++透视图（此处没有显示），则也可以管理Outline视图。这对于大型项目非常有帮助。

Eclipse更好地集成了编辑和编译过程。如果产生编译错误，则会在编辑器中清楚地标记出这些错误。可以使用Vim编辑器完成该操作，与IDE相比，本书的两位作者都倾向于使用Vim编辑器，但是IDE可以更好地执行该操作。

另一方面，可以看到与大多数IDE一样，Eclipse在屏幕（实际上是本书中的页面）上占据了大量的空间。无论使用多少，Outline视图都会占据屏幕中的宝贵空间。诚然，可以通过单击右上角的✖按钮来隐藏Outline视图以回收一些空间（如果需要恢复显示该视图，可以选择Window→Show View→Outline），也可以将选项卡拖动到Eclipse窗口中的不同位置。但是一般来说，可能很难较好地利用Eclipse中的屏幕空间。

记住，始终可以直接在DDD的控制台中执行GDB命令。因此，你可以灵活地以最为方便的方法执行调试命令，有时是通过DDD界面，而有时则是通过GDB命令行。在本书的不同部分中，将看到可以使用GDB执行大量操作，这些操作可以简化调试工作。

通过对比可以得知，GDB对于Eclipse用户来说基本是透明的，虽然过去的说法“无知即是快乐”通常也适用，但是这种透明性意味着你将不能方便地使用可以通过直接利用GDB执行的节省劳动量的操作。在编写本书时，心意坚决的用户仍然可以通过如下方法直接访问GDB：在调试透视图中单击GDB线程，然后使用控制台。但是这种方法缺少GDB提示符。然而，这种“未归档的功能”在将来的版本中可能不再存在。

5. GUI的优点

DDD和Eclipse提供的GUI界面比GDB提供的GUI界面的外观更加形象，并且使用起来也更方便。例如，倘若不再需要在`get_args()`的第16行处暂停执行，也就是希望清除该位置中的断点，在GDB中，可输入如下命令来清除断点。

```
(gdb) clear 16
```

然而，为了执行该操作，需要记住断点所在的行号（如果同时有许多有效的断点，记住这些行号就不容易）。可以使用GDB的`info break`命令列出所有断点，来重新记住所有的行号，但是这增加了一定的工作量，并且会分散你查找程序错误的注意力。

在DDD中，你的任务将会简单很多：为了清除断点，只需要单击所需行中的停止符号，然后单击Clear按钮，停止符号将会消失，表明已经清除该断点。

在Eclipse中，可以进入Breakpoint视图，突出显示需要删除的断点，然后将鼠标光标移动到灰色的✖按钮上，该按钮表示删除所选择断点的操作，如图1-4所示。也可以右击源代码窗口中的蓝色断点符号并选择Toggle Breakpoint命令。

判断何种GUI可以更为有效地执行清除断点工作的一项任务是单步调试代码。与GDB相比，

使用DDD或Eclipse可以更加方便而舒适地执行清除断点的工作，因为可以在GUI的源代码窗口中监视在代码中的移动。通过箭头指示源代码中将要执行的下一个代码行，如图1-5中的DDD源代码窗口所示。在Eclipse中，以绿色突出显示要执行的下一个代码行。因此，可以一眼看出目前相对于其他程序语句所在的位置。

6. GDB的优点

综上所述，与基于文本的GDB相比，这些GUI有许多优点。然而，根据该示例就笼统地断定GUI好于GDB并不见得正确。

成长过程中一直使用GUI完成联机执行的一切事务的年轻程序员，自然更喜欢使用GUI而非GDB，很多年纪大些的程序员也是如此。但是，GDB也有一些突出的优点。

- GDB的启动速度比DDD快许多，当只需要快速检查代码中的某项内容时，这就是很重要的优点。在与Eclipse比较时，这种启动时间上的优势更明显。
- 在某些情况下，需要从公共终端通过SSH（或远程登录）远程执行调试。如果没有安装X11，就完全不能使用GUI，即使有X11，GUI的屏幕刷新操作也会非常缓慢。
- 当调试彼此之间协同操作的多个程序时（例如，网络化环境中的客户/服务器程序），就需要针对每个程序打开独立的调试窗口。与DDD相比，Eclipse中的情况稍好，因为Eclipse允许在同一个窗口中同时调试两个程序，但是这也会带来前面提及的空间问题。因此，与GUI占据大量屏幕空间相比，GDB占据的可视空间少是很重要的优点。
- 如果正在调试的程序有GUI，并且使用的是基于GUI的调试器（如DDD），则两者之间就可能产生冲突。其中某个程序的GUI事件（按键、鼠标单击和鼠标移动），可能会干扰另一个程序的GUI事件，并且被调试的程序在调试器下运行时可能具有与独立运行时不同的行为。这可能会使查找程序错误变得相当复杂。

相比于GUI中便利的鼠标操作，GDB所需的大量输入操作显得十分麻烦，但必须注意的是GDB也包括一些减少输入量的方法，从而使其基于文本的特性更容易被用户接受。本章前面提及，GDB的大多数命令都有简短的缩写方式，大多数人都使用这种缩写方式而不是完整的命令名。同样，使用Ctrl+P和Ctrl+N组合键可以浏览以前的命令并在需要时编辑这些命令。只需要单击ENTER键即可重复发出上一个命令（在重复执行`next`命令一次一行地单步调试代码时，这种方法就非常有用）；此外还有`define`命令，用户可以定义缩写和宏。第2章和第3章将详细介绍这些功能。

7. 总结：每种工具都有其价值

我们认为GDB和GUI都是非常重要的工具，并且本书将同时给出GDB、DDD和Eclipse的相关示例。我们将始终使用GDB切入所有主题，因为GDB具有这些工具所共有的特性，然后再将讨论扩展到GUI。

1.4.2　折中方法

从版本6.1以来，GDB已经以名为TUI（Terminal User Interface，终端用户界面）的模式提供了基于文本交互和图形用户交互之间的折中方法。在这一模式中，GDB将终端屏幕划分为类似于

DDD的源文本窗口和控制台的多个子窗口：可以在类似于源文本窗口的子窗口中跟踪程序执行的进展过程，同时在类似于控制台的子窗口中发出GDB命令。也可以使用另一个程序CGDB，它也提供了类似的功能。

1. 处于TUI模式的GDB

为了以TUI模式运行GDB，可以在调用GDB时在命令行上指定`-tui`选项，或者处于非TUI模式时在GDB中使用Ctrl+X+A组合键。如果当前处于TUI模式，后一种命令方式就会使你离开TUI模式。

在TUI模式中，GDB窗口划分为两个子窗口，一个用于输入GDB命令，而另一个用于查看源代码。假设针对*insert_sort*以TUI模式启动GDB，然后执行几个调试命令，GDB屏幕就可能如下所示：

```
    11
    12      void get_args(int ac, char **av)
    13      {  int i;
    14
    15         num_inputs = ac - 1;
*   16         for (i = 0; i < num_inputs; i++)
 >  17            x[i] = atoi(av[i+1]);
    18      }
    19
    20      void scoot_over(int jj)
    21      {  int k;
    22
    23            for (k = num_y-1; k > jj; k++)
 File: ins.c    Procedure: get_args    Line: 17      pc: 0x80484b8
-------------------------------------------------------------------------
(gdb) break 16
Breakpoint 1 at 0x804849f: file ins.c, line 16.
(gdb) run 12 5 6
Starting program: /debug/insert_sort 12 5 6

Breakpoint 1, get_args (ac=4, av=0xbffff094) at ins.c:16
(gdb) next
(gdb)
```

如果正在使用GDB但没有使用TUI模式，则位于下方的子窗口确切地显示你将看到的内容。此处，该子窗口显示如下内容。

- 发出一条`break`命令，在当前源文件的第16行处设置断点。
- 执行`run`命令运行程序，并且向该程序传递命令行参数12、5和6。在此之后，调试器在指定的断点处停止执行（后面将介绍`run`和其他GDB命令）。GDB会提醒用户断点位于*ins.c*的第16行，并且通知该源代码行的机器代码驻留在内存地址`0x804849f`中。

- 发出next命令，步进到下一个代码行，即第17行。

上方的子窗口提供了一些额外的、形象的帮助信息。此处，与DDD和Eclipse相同，TUI显示了当前执行的代码行上下的其他源代码。这样就可以更方便地查看当前位于代码中的位置。分别使用星号和>符号指示断点和当前执行的代码行，这两个符号分别类似于DDD的停止符号和绿色箭头。

通过使用上下方向键进行滚动来移动到代码的其他部分。如果没有处于TUI模式中，就可以使用箭头键来浏览以前的GDB命令，从而修改或重复执行这些命令。在TUI模式中，箭头键用于滚动源代码子窗口，可以使用Ctrl+P和Ctrl+N组合键来浏览以前的GDB命令。同样，在TUI模式中，可以使用GDB的list命令更改源代码子窗口中显示的代码区域。在操作多个源文件的情况下，该功能就非常有用。

通过使用GDB的TUI模式和它的输入快捷方式，可以在不受GUI不利方面影响的情况下用到GUI的许多优秀功能。然而需要注意的是，在一些情况下TUI可能不能按照用户所需要的方式运作，这时就需要寻找相应的解决方案。

2. CGDB

另一个可用的GDB界面是CGDB，该界面可以从*http://cgdb.sourceforge.net/*获得。CGDB也提供了一种基于文本的方法与GUI方法之间的折中方案。与GUI类似，CGDB起GDB前端的作用。CGDB从概念上类似于基于终端的TUI，但它更吸引人，因为是彩色界面，而且可以浏览源代码子窗口，并直接在子窗口中设置断点。CGDB处理屏幕刷新的能力似乎也比GDB/TUI强。

下面是CGDB的几个基本命令与约定。

- 按下ESC键可以从命令窗口转到源代码窗口，按下i键返回。
- 当光标在源代码窗口中时，可以用键头键或功能类似于vi的键（j表示向下，k表示向上，/表示查找）在源代码窗口中随意移动。
- 要执行的下一行用箭头标记。
- 为了在通过光标突出显示的当前代码行上设置断点，只要按下空格键即可。
- 断点行的行号用红色突出显示。

1.5 主要调试器操作

本节概述调试器提供的主要操作类型。

1.5.1 单步调试源代码

前面介绍过，在GDB中是用run命令运行程序，在DDD中是单击Run按钮。在后面介绍详细内容时，可以看到Eclipse运行程序的方式也是类似的。

也可以安排程序在某个地方暂停执行，以便检查变量的值，寻求程序错误所在位置的线索。下面是可用来暂停程序执行的一些方法。

- 断点

正如前面所提到的，调试工具会在指定断点处暂停程序的执行。在GDB中是通过break命令

及源代码行号完成的，在DDD中是在相关代码行的任意空白处右击并选择Set Breakpoint来完成的，在Eclipse中是在代码行左边的页边空白处双击完成的。

- 单步调试

前面提到过，GDB的`next`命令让GDB执行下一行，然后暂停。`step`命令的作用与此类似，只是在函数调用时`step`命令会进入函数，而`next`导致下一次暂停出现在调用函数之后。DDD有对应的Next和Step菜单项，而Eclipse是通过Step Over和Step Into图标完成这一功能的。

- 恢复操作

在GDB中，`continue`命令通知调试器恢复执行，直到遇到新断点为止。DDD中有一个对应的菜单项，Eclipse中有一个Resume图标，都能完成这一功能。

- 临时断点

在GDB中，`tbreak`命令与`break`相似，但是这一命令设置的断点在首次到达该指定行后就不再有效，只使用一次。在DDD中临时断点的设置方式为：在源文本窗口中要设置断点的代码行的任意空白处右击，然后选择Set Temporary Breakpoint。在Eclipse中，突出显示源代码窗口中要设置断点的代码行，然后右击并选择Run to Line。

GDB中还有创建特殊类型的一次性断点的命令：`until`和`finish`。DDD的命令工具中有对应的Until和Finish项，Eclipse中有Step Return。这些内容将在第2章讨论。

程序执行的典型调试模式如下（以GDB为例）：单击一个断点后，通过GDB的`next`命令一次移动一行代码，或通过`step`命令单步调试一段时间，以便仔细检查靠近断点处的程序状态和行为。做完这些操作后，可以用`continue`命令让调试器继续执行程序，直到遇到下一个断点为止，其间不需要暂停。

1.5.2　检查变量

当调试器暂停了程序的执行后，可以执行一些命令来显示程序变量的值。这些变量可以是局部变量、全局变量、数组的元素和C语言的`struct`、C++类中的成员变量等。如果发现某个变量有一个出乎意料的值，那往往是找出某个程序错误的位置和性质的重要线索。DDD甚至可以用图来表示数组，从而直观地表达数组中的可疑值或者发生的某种趋势。

最基本的变量显示类型是仅仅输出当前值。例如，假设在*ins.c*中的函数`insert()`的第37行处设置了一个断点。（同样，完整的源代码将在1.7节中提供，但目前不需要关注细节。）当到达这一行时，可以查看该函数中局部变量`j`的值。在GDB中使用`print`命令输出当前值。

```
(gdb) print j
```

在DDD中输出当前值的方法更简单：只要将鼠标指针在源文本窗口中`j`的任一实例上移动，等1~2秒后就会在一个靠近鼠标指针的小黄框（称为值提示）中显示`j`的值。图1-5显示了被查看的变量`new_y`的值。Eclipse的显示方式也是如此，如图1-6所示，其中查询了`num_y`的值。

第2章将会介绍，在GDB或DDD中，也可以安排连续显示变量，这样就不必反复要求查看值。DDD有一个特别优秀的功能，可以显示链表、树以及其他包含指针的数据结构：单击这种结构中

任一节点的向外链接即可找到下一个节点。

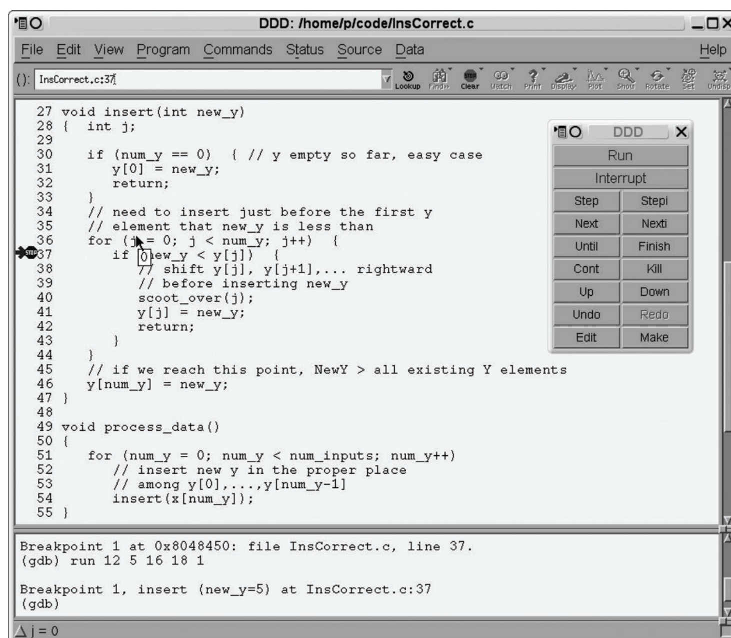


图1-5　在DDD中查看变量

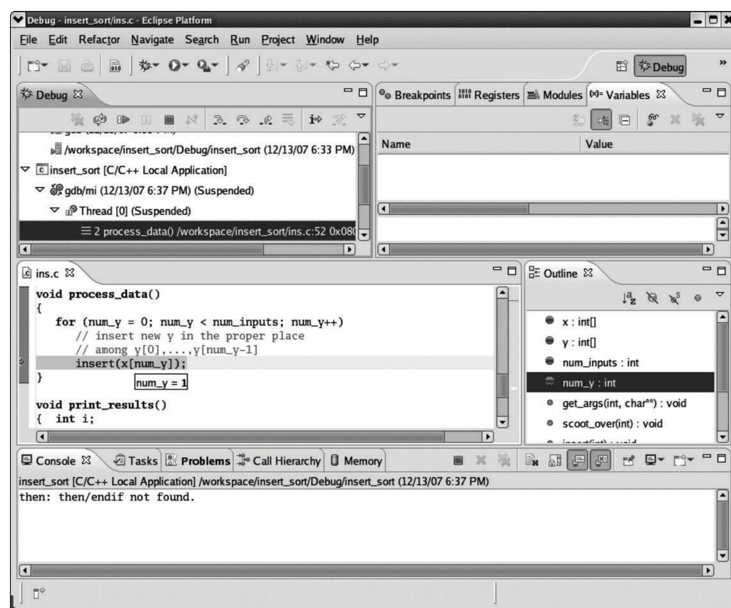


图1-6　在Eclipse中查看变量

1.5.3 在GDB、DDD和Eclipse中设置监视点以应对变量值的改变

监视点结合了断点和变量检查的概念。最基本形式的监视点通知调试器，每当指定变量的值发生变化时都暂停程序的执行。

例如，在程序执行期间，假设要在变量z改变值时查看程序的状态。在GDB中，可以执行如下命令。

```
(gdb) watch z
```

当运行程序时，每当z的值发生变化，GDB都会暂停执行。在DDD中，通过在源文本窗口中单击z的任一实例来设置监视点，然后单击DDD窗口上方的Watch图标。

更好的方法是，可以基于条件表达式来设置监视点。例如，假设要查找程序执行期间z的值大于28的第一个位置，可以通过设置一个基于表达式（z>28）的监视点来完成这件事。在GDB中，输入：

```
(gdb) watch (z > 28)
```

在DDD中，则是在DDD的控制台中执行这一命令。你一定记得，C语言中表达式（z>28）的结果是布尔类型，计算得到的值为*true*或*false*，其中*false*用0表示，*true*用任意非零整数值表示，通常用1表示。当z首次变成大于28的值时，表达式（z>28）会从0变成1，GDB暂停程序的执行。

在Eclipse中设置监视点的方法为：在源代码窗口中点击鼠标右键，选择Add a Watch Expression，然后在对话框中填写适当的表达式。

监视点对局部变量的用途一般没有对作用域更宽的变量的用途大，因为一旦变量超出作用域（即当在其中定义变量的函数结束时），在局部变量上设置的监视点就会被取消。然而，main()中的局部变量显然是一个例外，因为这样的变量只有程序执行完时才会被释放。

1.5.4 上下移动调用栈

在函数调用期间，与调用关联的运行时信息存储在称为栈帧（stack frame）的内存区域中。帧中包含函数的局部变量的值，函数的形参，以及调用该函数的位置的记录。每次发生函数调用时，都会创建一个新帧，并将其推到一个系统维护的栈上；栈最上方的帧表示当前正在执行的函数，当函数退出时，这个帧被弹出栈，并且被释放。

例如，在insert()函数中暂停示例程序*insert_sort*的执行。当前栈帧中的数据会指出，你通过在一个恰好位于process_data()函数（该函数调用insert()）中的特定位置进行的函数调用到达了这个帧。这个帧也会存储insert()的唯一局部变量的当前值，稍后你会发现这个值为j。

其他活动函数调用的栈帧将包含类似的信息，如果你愿意，也可以查看它们。例如，尽管程序的执行当前位于insert()内，但是你也可能希望查看调用栈中以前的帧，即查看process_data()的帧。在GDB中可以用如下命令查看以前的帧。

```
(gdb) frame 1
```

当执行GDB的`frame`命令时，当前正在执行的函数的帧被编号为0，其父帧（即该函数的调用者的栈帧）被编号为1，父帧的父帧被编号为2，以此类推。GDB的`up`命令将你带到调用栈中的下一个父帧（例如，从帧0到帧1），`down`则引向相反方向。这样的操作非常有用，因为根据以前的一部分栈帧中的局部变量的值，可能发现程序错误的线索。

遍历调用栈不会修改执行路径（在本例中，要执行的*insert_sort*的下一行仍然会是`insert()`中的当前行），但是它确实允许查看帧的祖先帧，因此可以检查通向当前帧的各次函数调用的局部变量的值。同样，这可以提示你从何处查找程序错误。

GDB的`backtrace`命令会显示整个栈，即当前存在的所有帧的集合。

DDD中的类似操作通过Status→Backtrace完成，这样做会弹出一个显示所有帧的窗口，然后可以单击要查看的任何一个帧。DDD界面也有Up和Down按钮，单击这两个按钮可以执行GDB的`up`和`down`命令。

在Eclipse中，栈在Debug透视图本身中连续可见。在图1-7中，查看左上角的Debug选项卡，可以看出我们目前在函数`get_args()`的第2帧中，这个函数是从`main()`中的第1帧调用的。突出显示的是源代码窗口中显示的那一帧，因此可以通过在调用栈中单击来显示任一帧。

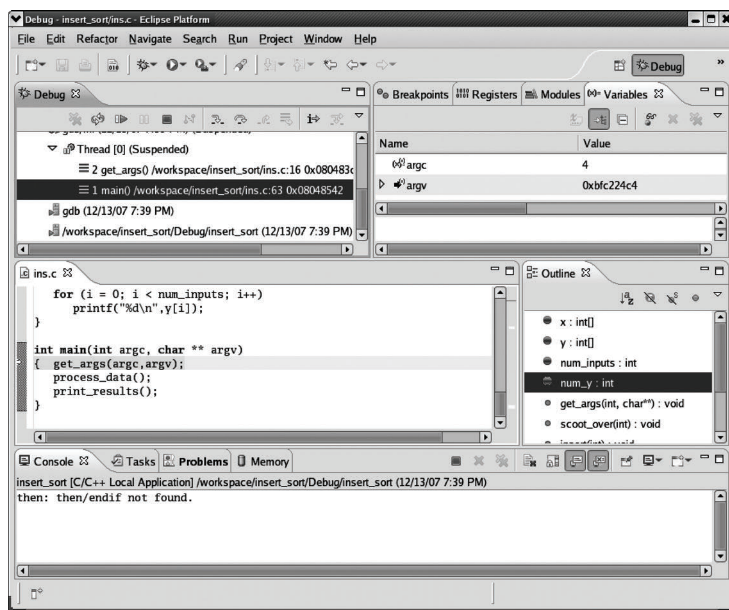


图1-7 在Eclipse的栈中移动

1.6 联机帮助

在GDB中，可以通过`help`命令访问文档。例如，

```
(gdb) help breakpoints
```

将显示关于断点的文档。不带参数的GDB命令`help`可显示一个菜单，列出可用来作为`help`的参数的命令类别。

在DDD和Eclipse中，可通过单击Help获得大量信息。

1.7　初涉调试会话

现在我们将提供一个完整的调试会话。前面说过，示例程序在源文件*ins.c*中，它实现插入排序。当然，这种排序方法效率不高，但是其代码比较简单，有利于说明调试操作，代码如下。

```
//
// insertion sort, several errors
//
// usage:  insert_sort num1 num2 num3 ..., where the numi are the numbers to
// be sorted

int x[10],  // input array
    y[10],  // workspace array
    num_inputs,  // length of input array
    num_y = 0;  // current number of elements in y

void get_args(int ac, char **av)
{  int i;

   num_inputs = ac - 1;
   for (i = 0; i < num_inputs; i++)
      x[i] = atoi(av[i+1]);
}

void scoot_over(int jj)
{  int k;

   for (k = num_y-1; k > jj; k++)
      y[k] = y[k-1];
}

void insert(int new_y)
{  int j;

   if (num_y = 0)  { // y empty so far, easy case
      y[0] = new_y;
      return;
   }
   // need to insert just before the first y
   // element that new_y is less than
```

```
      for (j = 0; j < num_y; j++)  {
         if (new_y < y[j])  {
            // shift y[j], y[j+1],... rightward
            // before inserting new_y
            scoot_over(j);
            y[j] = new_y;
            return;
         }
      }
   }

   void process_data()
   {
      for (num_y = 0; num_y < num_inputs; num_y++)
         // insert new y in the proper place
         // among y[0],...,y[num_y-1]
         insert(x[num_y]);
   }

   void print_results()
   {  int i;

      for (i = 0; i < num_inputs; i++)
         printf("%d\n",y[i]);
   }

   int main(int argc, char ** argv)
   {  get_args(argc,argv);
      process_data();
      print_results();
   }
```

下面是这段程序的一个伪码描述。用`call`语句表示函数调用，各个函数的伪码在这些调用下面缩进显示。

```
call main():
   set y array to empty
   call get_args():
      get num_inputs numbers x[i] from command line
   call process_data():
      for i = 1 to num_inputs
         call insert(x[i]):
            new_y = x[i]
            find first y[j] for which new_y < y[j]
            call scoot_over(j):
```

```
            shift y[j], y[j+1], ... to right,
               to make room for new_y
      set y[j] = new_y
```

让我们编译并运行代码。

```
$ gcc -g -Wall -o insert_sort ins.c
```

注意，在GCC中可以用-g选项让编译器将符号表（即对应于程序的变量和代码行的内存地址列表）保存在生成的可执行文件（这里是*insert_sort*）中。这是一个绝对必要的步骤，这样才能在调试会话过程中引用源代码中的变量名和行号。如果没有这一步（要是使用的编译器不是GCC，则必须做一些类似的操作），就不能要求调试器"在第30行处停止"或者"输出x的值"。

现在让我们运行该程序。根据1.3.3节介绍的从简单工作开始调试的原则，首先尝试排序一个只有两个数的列表。

```
$ insert_sort 12 5
(execution halted by user hitting ctrl-C)
```

该程序没有终止，也没有输出任何内容。它显然进入了一个无限循环，必须按下Ctrl+C组合键来终止这个循环。毫无疑问，肯定有什么地方出了问题。

在下面几节中，我们首先用GDB提供一个调试会话来调试这个有错误的程序，然后讨论如何用DDD和Eclipse进行这种调试。

1.7.1　GDB方法

为了跟踪第一个程序错误，在GDB中执行这个程序，并在按Ctrl+C组合键挂起程序之前让它运行一会儿。然后看看这时停留在何处。用这种方式可以确定无限循环的位置。

首先，对*insert_sort*启动GDB调试器。

```
$ gdb insert_sort -tui
```

这时屏幕显示如下。

```
    63      {  get_args(argc,argv);
    64         process_data();
    65         print_results();
    66      }
    67
    68
    69
 File: ins.c    Procedure: ??    Line: ??      pc: ??
-----------------------------------------------------------------------
(gdb)
```

上面的子窗口显示了部分源代码，在下面的子窗口中有等待输入命令的GDB提示符，还有一

条GDB欢迎消息，为了简化起见，我们省略了这条消息。

如果在调用GDB时没有请求TUI模式，那么只会收到欢迎消息和GDB提示符，不会出现上面的程序源代码子窗口。可以用GDB的命令Ctrl+X+A组合键进入TUI模式。这个命令用来打开或关闭TUI模式。如果你愿意，可以用该命令临时离开TUI模式，从而更方便地阅读GDB的联机帮助，或者在同一个屏幕上查看用过的更多GDB命令。

现在从GDB中执行`run`命令以及程序的命令行参数来运行该程序，然后按Ctrl+C组合键挂起它。屏幕现在如下所示。

```
    46
    47 void process_data()
    48 {
    49 for (num_y = 0; num_y < num_inputs; num_y++)
    50 // insert new y in the proper place
    51 // among y[0],...,y[num_y-1]
  > 52 insert(x[num_y]);
    53 }
    54
    55 void print_results()
    56 { int i;
    57
    58 for (i = 0; i < num_inputs; i++)
    59 printf("%d\n",y[i]);
    60 } .
 File: ins.c Procedure: process_data Line: 52 pc: 0x8048483
--------------------------------------------------------------------------
(gdb) run 12 5
Starting program: /debug/insert_sort 12 5

Program received signal SIGINT, Interrupt.
0x08048483 in process_data () at ins.c:52
(gdb)
```

该屏幕表明，当停止程序时，*insert_sort*在函数`process_data()`中，即将执行源文件*ins.c*的第52行。

我们随机按下Ctrl+C组合键，并在代码中的任意位置停止。有时最好挂起并重启停止了两到三次响应的程序，方法是在两次按下Ctrl+C组合键之间执行`continue`命令，以便查看每次是在何处停止的。

现在，第52行是从第49行开始的循环的一部分。这个循环是否为无限循环呢？这个循环似乎不应当无限地循环，但是确认原则表明，应该验证一下，不要想当然。如果因为你不知何故没有正确地设置变量`num_y`的上边界，导致循环没有终止，那么当程序运行了一段时间后，`num_y`的值将相当大。是吧？（同样，看上去似乎不会，但是需要确认一下。）下面我们查看一下GDB输出的`num_y`值。

```
(gdb) print num_y
$1 = 1
```

这一查询的输出表明num_y的值为1。$1标签意味着这是你要求GDB输出的第一个值。（$1、$2、$3等表示的值统称为调试会话的值历史。后面的章节中你会看到，会话的值历史非常有用。）因此我们似乎仅在该循环的第二个迭代上，即第49行。如果这个循环是无限循环，那么到这时为止应该远不止是第二次迭代。

那么，让我们仔细看一下当num_y为1时发生的事情。通知GDB在循环的第二次迭代期间，在第49行处的insert()中停止，以便查看情况，尝试找出程序这时在这个位置出了什么问题。

```
(gdb) break 30
Breakpoint 1 at 0x80483fc: file ins.c, line 30.
(gdb) condition 1 num_y==1
```

第一个命令在第30行处（即在insert()的开头）放置一个断点。另外，可以通过命令break insert指定这个断点，即在insert()的第一行处中断（这里是第30行）。后一种形式有一个优点：如果修改了程序代码，使得函数insert()不再在*ins.c*的第30行处开始，那么如果用函数名指定断点，而不是用行号指定，则断点仍然有效。

break命令一般会使得每次程序执行到指定行时都会暂停。然而，这里的第二个命令condition 1 num_y==1使得该断点成为有条件的断点：只有当满足条件num_y==1时，GDB才会在断点1处暂停程序的执行。

注意，与接受行号（或函数名）的break命令不同，condition接受断点号。总是可以用命令info break来查询要查找的断点的编号。（该命令也提供了其他有用信息，比如到目前为止遇到各个断点的次数。）

用break if可以将break和condition命令组合成一个步骤，如下所示。

```
(gdb) break 30 if num_y==1
```

然后用run命令再次运行程序。如果要重用老的命令行参数，就不必再次指定命令行参数。这里就是这种情况，可以简单地输入run。由于程序已经在运行，因此GDB询问是否希望重新从头开始，请回答“是”。

屏幕这时将如下所示。

```
   24            y[k] = y[k-1];
   25      }
   26
   27      void insert(int new_y)
   28      {  int j;
   29
*> 30         if (num_y = 0)  { // y empty so far, easy case
   31            y[0] = new_y;
   32            return;
```

```
     33         }
     34         // need to insert just before the first y
     35         // element that new_y is less than
     36         for (j = 0; j < num_y; j++)  {
     37            if (new_y < y[j])  {
     38               // shift y[j], y[j+1],... rightward            .
  File: ins.c    Procedure: insert    Line: 30      pc: 0x80483fc
--------------------------------------------------------------------------
(gdb) condition 1 num_y==1
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
Starting program: /debug/insert_sort 12 5

Breakpoint 1, insert (new_y=5) at ins.c:30
(gdb)
```

我们再次应用确认原则：因为num_y为1，所以应跳过第31行，直接执行第36行。但是我们需要确认这一点，因此执行next命令来继续执行下一行。

```
     24            y[k] = y[k-1];
     25      }
     26
     27      void insert(int new_y)
     28      {  int j;
     29
  *  30         if (num_y = 0)  { // y empty so far, easy case
     31            y[0] = new_y;
     32            return;
     33         }
     34         // need to insert just before the first y
     35         // element that new_y is less than
   > 36         for (j = 0; j < num_y; j++)  {
     37            if (new_y < y[j])  {
     38               // shift y[j], y[j+1],... rightward            .
  File: ins.c    Procedure: insert    Line: 36      pc: 0x8048406
--------------------------------------------------------------------------
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
Starting program: /debug/insert_sort 12 5

Breakpoint 1, insert (new_y=5) at ins.c:30
(gdb) next
(gdb)
```

上面的子窗口中的箭头现在位于第36行，因此确认了我们的预料，我们确实跳过了第31行。现在让我们继续单步调试程序，依次确认关于这段代码的假设。由于现在位于循环的开头，因此再执行几次next命令来逐行查看循环的进展。

```
     39            // before inserting new_y
     40            scoot_over(j);
     41            y[j] = new_y;
     42            return;
     43         }
     44      }
   > 45   }
     46
     47   void process_data()
     48   {
     49      for (num_y = 0; num_y < num_inputs; num_y++)
     50         // insert new y in the proper place
     51         // among y[0],...,y[num_y-1]
     52         insert(x[num_y]);
     53   }                                                    .
 File: ins.c    Procedure: insert    Line: 45      pc: 0x804844d
--------------------------------------------------------------------------
The program being debugged has been started already.
Start it from the beginning? (y or n)
Starting program: /debug/insert_sort 12 5

Breakpoint 1, insert (new_y=5) at ins.c:30
(gdb) next
(gdb) next
(gdb)
```

看一下上面的子窗口中箭头现在位于何处——我们直接从第37行跳到了第45行！这种情况让人大吃一惊，这个循环一次迭代都没有执行。不过要记住，这种惊讶是好事，因为它们提供了关于程序错误在何处的线索。

该循环在第36行处根本不执行迭代的唯一原因是：即使当j为0时，第36行的条件j<num_y也不成立。然而你知道（至少你认为你知道）num_y为1，因为在该断点上施加了条件num_y==1后，你现在位于该函数中。同样，还没有确认这一点。现在检查一下。

```
(gdb) print num_y
$2 = 0
```

毫无疑问，虽然进入insert()时不满足num_y==1这一条件，但是显然从那时起num_y已经改变。当进入这个函数后，num_y变成了0。但是如何变的呢？

前面说过，确认原则不会告诉你程序错误是什么，但是它提供了程序错误可能位于何处的线

索。在本例中，现在发现了程序错误的位置在第30行和第36行之间的某处。可以进一步缩小其范围，因为你看到第31行到第33行被跳过了，第34行到第35行是注释。换言之，要么在第30行，要么在第36行，num_y中的值发生了神秘的变化。

在进行短暂的休息（这往往是最佳调试策略！）之后，我们突然意识到这个问题是C程序员新手常犯的一个典型错误（有经验的程序员有时也会狼狈地犯这个错误）：在第30行我们用=，而不是==，把一次相等测试变成了一个赋值操作。

明白了吗？因为这个原因，所以产生了无限循环。第30行的错误产生了一种无休止的拉锯式情况，第49行的num_y++部分让num_y反复地从0递增到1，而第30行的错误反复地将该变量的值又设置为0。

因此我们修复了这个比较丢脸的程序错误，重新编译，并再次运行程序。

```
$ insert_sort 12 5
5
0
```

虽然我们不再有无限循环了，但是仍然没有得到正确的输出。

从前面描述程序功能的伪码中可以看出：最初数组y应该是空的，循环在第49行的第一次迭代应该将12放到y[0]中，然后在第二次迭代中12应移动一个数组位置，为插入5腾出空间。然而，事实是5替换了12。

问题出在第二个数字（5）上，因此应该再次聚焦第二次迭代。因为我们明智地选择了留在GDB会话中，而没有在发现和修复第一个程序错误后退出GDB，所以我们以前设置的断点及其条件现在仍然有效。因此只要再次运行程序，当程序开始处理第二个输入时停止。

```
      24            y[k] = y[k-1];
      25      }
      26
      27      void insert(int new_y)
      28      {  int j;
      29
   *> 30         if (num_y == 0)  { // y empty so far, easy case
      31            y[0] = new_y;
      32            return;
      33         }
      34         // need to insert just before the first y
      35         // element that new_y is less than
      36         for (j = 0; j < num_y; j++)  {
      37            if (new_y < y[j])  {
      38               // shift y[j], y[j+1],... rightward             .
 File: ins.c    Procedure: insert    Line: 30     pc: 0x80483fc
--------------------------------------------------------------------------
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

```
`/debug/insert_sort' has changed; re-reading symbols.
Starting program: /debug/insert_sort 12 5

Breakpoint 1, insert (new_y=5) at ins.c:30
(gdb)
```

注意如下这行代码。

```
`/debug/insert_sort' has changed; re-reading symbols.
```

这表示GDB发现我们重新编译了程序，在运行程序之前自动重新加载了新的二分表和新的符号表。

我们在重新编译程序之前仍然不必退出GDB。这样做之所以提供了很大的方便，有几个原因。第一，不需要重新指出命令行参数，只要键入run重新运行程序即可。其次，GDB保留了你设置的断点，因此不需要再次键入它。虽然这里只有一个断点，但很多情况下会有多个断点，因此这样省了不少事。这些便利减少了键入操作，更重要的是它们减少了对机械操作的分心，可以更好地将精力集中于实际调试上。

同样，在调试会话期间不要退出再重启文本编辑器，这件事也会分心而且浪费时间。只要将文本编辑器放在一个窗口中，GDB（或DDD）放在另一个窗口中，用第三个窗口调试程序即可。

现在让我们再次尝试单步调试代码。与以前一样，程序应该跳过第31行，但是与前面的情况不同的是，这次它有希望到达第37行。我们通过执行next命令两次来检查这一点。

```
   31            y[0] = new_y;
   32            return;
   33         }
   34         // need to insert just before the first y
   35         // element that new_y is less than
   36         for (j = 0; j < num_y; j++)  {
 > 37            if (new_y < y[j])  {
   38               // shift y[j], y[j+1],... rightward
   39               // before inserting new_y
   40               scoot_over(j);
   41               y[j] = new_y;
   42               return;
   43            }
   44         }
   45      }                                                    .
 File: ins.c    Procedure: insert    Line: 37      pc: 0x8048423
---------------------------------------------------------------------------
`/debug/insert_sort' has changed; re-reading symbols.
Starting program: /debug/insert_sort 12 5

Breakpoint 1, insert (new_y=5) at ins.c:30
(gdb) next
```

```
(gdb) next
(gdb)
```

我们真的到达了第37行。

这时，我们认为第37行的if中的条件应当成立，因为new_y应为5，并且第一次迭代结束y[0]应为12。GDB的输出确认了前一个假设，下面检查后一个假设。

```
(gdb) print y[0]
$3 = 12
```

这个假设也得到了确认，然后执行next命令，它将你带到第40行。我们预期函数scoot_over()将12移到下一个数组位置处，为5腾出空间。你应该查看它有没有完成这件事。这时面临一个重要选择。你可以再次执行next命令，使得GDB在第41行处停止；函数scoot_over()会被执行，但是GDB不会在该函数中停止。然而，如果你执行的是step命令，GDB就会在第23行停止，这样会允许在scoot_over()中单步调试。

采用1.3.3节介绍的自顶向下的调试方法，我们在第40行选择next命令，而不是step命令。当GDB在第41行停止时，可以看一下y值，检查函数有没有正确地完成其工作。如果确认了这一假设，就不必浪费时间查看对修复当前程序错误没有用处的函数scoot_over()的详细操作。如果没有能够确认，可以再次在调试器中运行该程序，并使用step进入函数，以便检查函数的详细操作，以期确定何处出了问题。

因此，当达到第40行时，键入next，产生：

```
   31            y[0] = new_y;
   32            return;
   33         }
   34         // need to insert just before the first y
   35         // element that new_y is less than
   36         for (j = 0; j < num_y; j++)  {
   37            if (new_y < y[j])  {
   38               // shift y[j], y[j+1],... rightward
   39               // before inserting new_y
   40               scoot_over(j);
 > 41               y[j] = new_y;
   42               return;
   43            }
   44         }
   45      }                                                          .
 File: ins.c    Procedure: insert    Line: 41      pc: 0x8048440
---------------------------------------------------------------------------
(gdb) next
(gdb) next
(gdb)
```

scoot_over()有没有正确地移动12？让我们看一下。

```
(gdb) print y
$4 = {12, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

显然没有。问题实际上出现在scoot_over()中。我们删除insert()开头的断点，并在scoot_over()中放置一个断点，同样采用一个可在第49行的第二次迭代时停止的条件。

```
(gdb) clear 30
Deleted breakpoint 1
(gdb) break 23
Breakpoint 2 at 0x80483c3: file ins.c, line 23.
(gdb) condition 2 num_y==1
```

现在再次运行程序。

```
   15         num_inputs = ac - 1;
   16         for (i = 0; i < num_inputs; i++)
   17         x[i] = atoi(av[i+1]);
   18      }
   19
   20      void scoot_over(int jj)
   21      {  int k;
   22
*> 23         for (k = num_y-1; k > jj; k++)
   24            y[k] = y[k-1];
   25      }
   26
   27      void insert(int new_y)
   28      {  int j;
   29
File: ins.c    Procedure: scoot_over    Line: 23      pc: 0x80483c3
--------------------------------------------------------------------------
(gdb) condition 2 num_y==1
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
Starting program: /debug/insert_sort 12 5

Breakpoint 2, scoot_over (jj=0) at ins.c:23
(gdb)
```

再次遵循确认原则：想一下你预期会发生什么情况，然后尝试确认确实发生了这种情况。在这个例子中，我们认为函数会将12移到数组y中的下一个位置，这意味着第23行的循环应当恰好通过一次迭代。让我们通过重复执行next命令来单步调试该程序，以便确认这种预期。

```
     15          num_inputs = ac - 1;
     16          for (i = 0; i < num_inputs; i++)
     17          x[i] = atoi(av[i+1]);
     18       }
     19
     20       void scoot_over(int jj)
     21       {  int k;
     22
   * 23          for (k = num_y-1; k > jj; k++)
     24             y[k] = y[k-1];
    > 25       }
     26
     27       void insert(int new_y)
     28       {  int j;
     29
 File: ins.c    Procedure: scoot_over    Line: 25      pc: 0x80483f1
---------------------------------------------------------------------------
The program being debugged has been started already.
Start it from the beginning? (y or n)
Starting program: /debug/insert_sort 12 5

Breakpoint 2, scoot_over (jj=0) at ins.c:23
(gdb) next
(gdb) next
(gdb)
```

这里我们再次惊讶地发现：我们现在位于第25行，甚至都没有碰到第24行——循环没有执行迭代，我们预期会执行的迭代一次也没有执行。显然第23行有一个程序错误。

正如前面那个出乎意料地没有对循环体执行一次迭代的循环，肯定是在循环的一开始就没有满足循环条件。这里是不是这种情况呢？第23行的循环条件是`k>jj`。我们从这一行中还知道了`k`的初始值是`num_y-1`，我们从断点条件知道后者的量为0。最后，GDB屏幕告诉我们，`jj`为0。因此当循环开始时条件`k>jj`没有得到满足。

因此，我们要么错误地指定了循环条件`k>jj`，要么错误初始化成了`k=num_y-1`。我们认为在循环的第一次也是唯一的一次迭代中，应将12从`y[0]`移到`y[1]`（即第24行应当在`k=1`时已经执行），我们意识到循环初始化错了，应该是`k=num_y`。

修复这个错误，重新编译程序，再次运行程序（在GDB之外）。

```
$ insert_sort 12 5
Segmentation fault
```

当运行程序试图访问不允许访问的内存时，就会发生**段错误**（第4章将详细介绍）。原因通常是由于数组索引超出了边界，或者采用了错误的指针值。段错误也可能由没有显式地包含指针或数组变量的内存引用产生。这种情况可以从C程序员常犯的另一个典型错误中看出，比如，忘记

在用按引用调用传递的函数形参中加上&，写成了：

```
scanf("%d",x);
```

而不是

```
scanf("%d",&x);
```

一般而言，GDB或DDD这样的调试工具的主要价值是使人验证编码假设的过程更方便，但是对于段错误，调试工具提供了额外、切实、即时的帮助：它指出在程序中何处出现了错误。

为了利用这一点，需要在GDB中运行*insert_sort*，并重现段错误。首先，删除断点。正如前面见到的，要做到这一点，需要给出断点的行号。你可能已经记住了行号，但是要查找起来也很容易：可以滚动TUI窗口（用上下箭头键），查找用星号标记的行，或者用GDB的`info break`命令来查找。然后用`clear`命令删除断点。

```
(gdb) clear 30
```

现在再次在GDB中运行程序。

```
    19
    20      void scoot_over(int jj)
    21      {  int k;
    22
    23         for (k = num_y; k > jj; k++)
  > 24            y[k] = y[k-1];
    25      }
    26
    27      void insert(int new_y)
    28      {  int j;
    29
    30         if (num_y == 0)  { // y empty so far, easy case
    31            y[0] = new_y;
 File: ins.c    Procedure: scoot_over    Line: 24      pc: 0x8048538
--------------------------------------------------------------------------
Start it from the beginning? (y or n)

`/debug/insert_sort' has changed; re-reading symbols.
Starting program: /debug/insert_sort 12 5

Program received signal SIGSEGV, Segmentation fault.
0x08048538 in scoot_over (jj=0) at ins.c:24
(gdb)
```

不出所料，GDB告诉了我们段错误发生的确切位置，即在第24行，肯定与一个数组索引有关，

也就是k。要么是k太大了，超过了y中的数组个数，要么k-1是负数。显然我们需要做的第一件事是确定k的值。

```
(gdb) print k
$4 = 584
```

可以看出，代码规定y只能有10个元素，因此k的值实际上远远超过了这一范围。我们现在必须跟踪原因。

首先，确定段错误发生时这个重要循环迭代在第49行。

```
(gdb) print num_y
$5 = 1
```

因此是在第二次迭代期间也就是第一次执行函数scoot_over()时发生了段错误。换言之，并不是前几次调用scoot_over()时第23行工作得很好而在以后调用时失败了。这行代码仍然有一些根本性的错误。由于只剩下唯一的候选语句

```
k++
```

（因为前面检查过这一行的其他两个部分），因此问题肯定出在这里。再休息片刻，醒醒脑，我们意识到原来这里应该是k--。

修复这行代码并再次重新编译运行程序。

```
$ insert_sort 12 5
5
12
```

现在已经有了进步！但是该程序对于较大的数据集是否运行正确呢？让我们尝试一下。

```
$ insert_sort 12 5 19 22 6 1
1
5
6
12
0
0
```

现在已经看到了成功的曙光。大多数数组被正确地排序。该列表中的第一个没有正确排序的数字是19，因此在第36行设置一个断点，这次采用条件new_y == 19。[1]

```
(gdb) b 36
Breakpoint 3 at 0x804840d: file ins.c, line 36.
(gdb) cond 3 new_y==19
```

① 从这时起使用命令的常用缩写。比如，b表示break，i b表示info break，cond表示condition，r表示run，n表示next，s表示step，c表示continue，p表示print，bt表示backtrace。

然后在GDB中运行程序（一定要使用相同的参数，`12 5 19 22 6 1`）。当遇到断点时，确认到目前为止数组y已经被正确地排序：

```
     31            y[0] = new_y;
     32            return;
     33         }
     34         // need to insert just before the first y
     35         // element that new_y is less than
  *> 36         for (j = 0; j < num_y; j++)  {
     37            if (new_y < y[j])  {
     38               // shift y[j], y[j+1],... rightward
     39               // before inserting new_y
     40               scoot_over(j);
     41               y[j] = new_y;
     42               return;
     43            }
 File: ins.c    Procedure: insert    Line: 36      pc: 0x8048564
--------------------------------------------------------------------------
Start it from the beginning? (y or n)

Starting program: /debug/insert_sort 12 5 19 22 6 1

Breakpoint 2, insert (new_y=19) at ins.c:36
   (gdb) p y
   $1 = {5, 12, 0, 0, 0, 0, 0, 0, 0, 0}
   (gdb)
```

到目前为止，一切正常。现在让我们尝试确定程序如何处理19。我们仍然一次一行代码进行调试。注意，因为19既不小于5，也不小于12，所以我们没有期望第37行的`if`语句中的条件成立。当遇到n几次以后，我们发现自己位于第45行上：

```
     35         // element that new_y is less than
  *  36         for (j = 0; j < num_y; j++)  {
     37            if (new_y < y[j])  {
     38               // shift y[j], y[j+1],... rightward
     39               // before inserting new_y
     40               scoot_over(j);
     41               y[j] = new_y;
     42               return;
     43            }
     44         }
   > 45      }
     46
     47      void process_data()
 File: ins.c    Procedure: insert    Line: 45      pc: 0x80485c4
```

```
------------------------------------------------------------------------
  (gdb) n
  (gdb) n
  (gdb) n
  (gdb) n
  (gdb) n
  (gdb)
```

我们在第45行上，打算退出循环，根本没有对19做任何事情！进一步检查结果表明，我们的代码没有覆盖一个重要情况，也就是new_y大于我们目前为止处理的元素的情况，第34行和第35行的注释还揭露了一个疏漏：

```
// need to insert just before the first y
// element that new_y is less than
```

为了处理这种情况，在第44行后面添加如下代码：

```
// one more case:  new_y > all existing y elements
y[num_y] = new_y;
```

然后重新编译并再次运行程序：

```
$ insert_sort 12 5 19 22 6 1
1
5
6
12
19
22
```

这是正确的输出，接下来的测试也给出了正确的结果。

1.7.2 同样的会话在DDD中的情况

本节介绍上面的GDB会话在DDD中是什么情况。当然不需要重复所有步骤，只要关注与GDB不同的内容即可。

DDD的启动与GDB相似。用GCC编译源代码，使用-g选项，然后键入

```
$ ddd insert_sort
```

从而调用DDD。在GDB中，通过run命令开始程序的执行，如果有命令行参数的话，就要加上命令行参数。在DDD中，单击Program→Run，然后将看到图1-8所示的屏幕。

这时弹出了Run窗口，列出了已经使用过的命令行参数。这里以前还没有参数集，如果有，可以通过单击其中任何一个来做出选择，也可以按这里所示键入一组新参数。然后单击Run按钮。

在GDB调试会话中，我们在调试器中运行了一会儿程序，然后用Ctrl+C组合键挂起它，以便研究一个明显的无限循环。在DDD中，我们则通过在命令工具中单击Interrupt来挂起程序。DDD屏幕现在如图1-9所示。因为DDD起GDB前端的作用，所以这个鼠标单击在GDB中被翻译为Ctrl+C操作，从控制台中可以看到这一点。

上面的GDB会话中的下一步是检查变量num_y。如前面1.5节中所示，在DDD中，通过在源窗口中num_y的任意实例上移动鼠标来检查该变量。

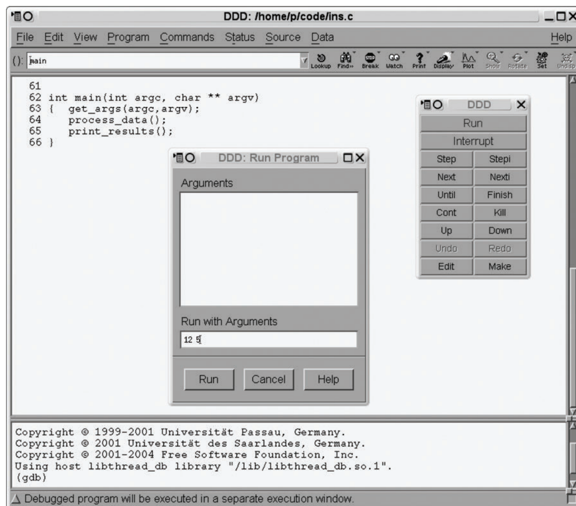


图1-8　DDD的Run命令

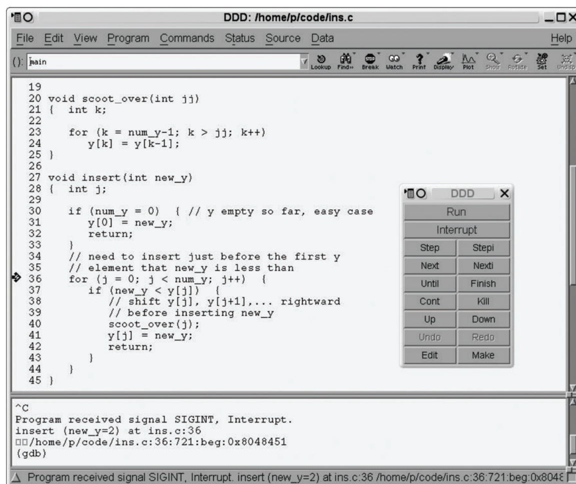


图1-9　中断后的屏幕

也可以用这种方式检查整个数组。例如，在GDB会话中的某一个位置，输出整个数组y。在

DDD中，只要将鼠标指针移到源窗口中y的任意实例上即可。如果在第30行上的表达式`y[j]`中的y上移动光标，屏幕会如图1-10所示。这一行附近会出现一个值提示框，显示y的内容。

在GDB会话中的下一个动作是在第30行设置一个断点。虽然我们已经解释了如何在DDD中设置断点，但是如果像本例中这样需要在断点上放置条件，该怎么办呢？可以通过右击断点行中的停止标记，然后选择Properties来完成这件事。这时会出弹出一个窗口，如图1-11所示。然后键入条件：`num_y==1`。

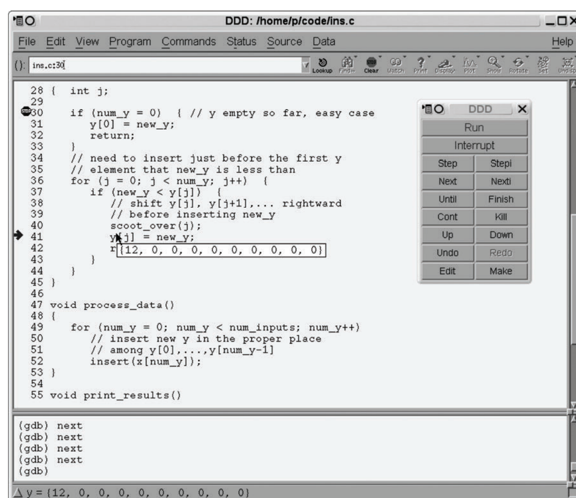


图1-10　检查数组

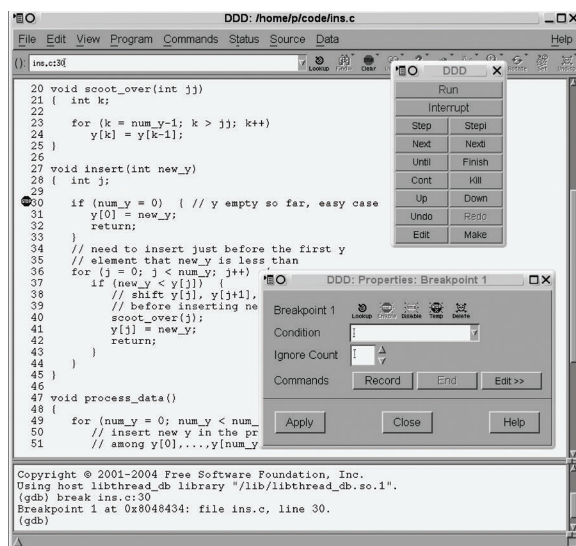


图1-11　在断点上施加一个条件

然后，为了重新运行程序，在命令工具中单击Run按钮。由于GDB的run命令没有参数，因此这个按钮用此前提供的最后一组参数运行程序。

在DDD中，与GDB的n和s命令对应的是命令工具中的Next和Step按钮。对应于GDB中的c命令的是Cont按钮。

本节的概述足够用DDD开始调试了。在以后的各章中将探索DDD的一些高级选项，比如非常有用的可视化显示复杂数据结构（如链表和二叉树）的功能。

1.7.3　Eclipse中的会话

现在让我们看一下上面的GDB会话在Eclipse中是什么情况。与在DDD中指出的一样，不需要重复所有步骤，我们直接将重点放在不同于GDB的内容上。

注意，Eclipse会相当讲究。虽然它提供了很多方式完成某种任务，但是如果没有严格遵循必需的步骤，你可能会发现除了重启部分调试过程之外，根本没有直观的解决方案。

这里我们假设已经创建了C/C++项目。[①]

当第一次运行或调试程序时，需要运行和调试配置文件。这些操作指定可执行文件的名称（以及它属于什么项目）、其命令行参数（如果有的话）、其特殊shell变量环境（如果有的话）、所选择的调试器，等等。*run*配置文件用来在调试器之外运行程序，而*debug*配置文件用来在调试内运行程序。一定要按这个顺序创建两个配置文件，如下所示。

(1) 选择Run → Open Run Dialog。

(2) 右击C/C++ Local Applications并选择New。

(3) 选择Main选项卡，并填写运行配置文件、项目及可执行文件名（Eclipse可能会提供一些提示），如果有终端I/O，选中Connect process input and ouput to a terminal框。

(4) 如果有命令行参数或特殊环境变量，单击Arguments或Environment选项卡，并填写所需设置。

(5) 选择Debugger选项卡以查看使用的是哪种调试器。虽然你可能不需要接触这一点，但是最好要知道，有一个底层调试器，或许是GDB。

(6) 单击Apply（如果看到这样的要求）和Close按钮来完成运行配置文件的创建。

(7) 通过选择Run→Open Debug Dialog来创建调试配置文件。Eclipse可能会重用你的运行配置文件中提供的信息，如图1-12所示；如果你愿意，也可以修改它。再次单击Apply（如果要求了）和Close按钮以完成调试配置文件的创建。

可以创建几个运行/调试配置文件，通常是用不同的命令行参数集。

为了启动调试会话，必须通过选择Window→Open Perspective→Debug来移到Debug透视图中。（有各种各样的快捷方式，留待读者去发现。）

当初次实际执行运行或调试动作时，同样要通过Run→Open Run Dialog或Run→Open Debug

① 由于本书关于调试，而不关于项目管理，因此这里不打算过多地介绍如何在Eclipse中创建和构建项目。然而，这里可以扼要说明一下创建项目的步骤：选择File→New→Project，选择C（或C++）项目，填写一个项目名，选择Executable → Finish。自动创建一个makefile文件。通过选择Project→Build Project来构建项目（即编译与连接）。

Dialog（根据情况而定），以指出使用哪个配置文件。然而，从那以后只要选择Run→Run或Run→Debug，都可以再次运行上一个调试配置文件。

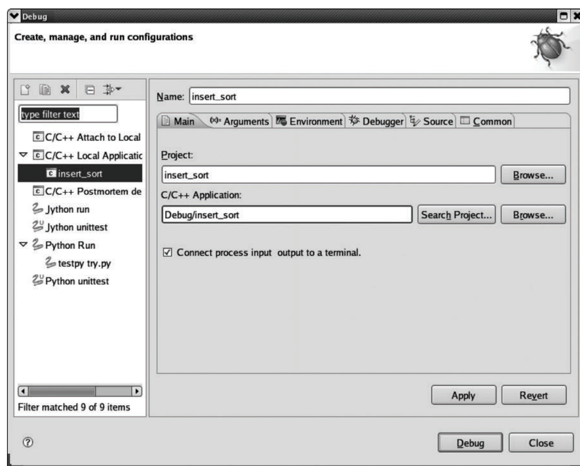


图1-12　Debug配置对话框

事实上，在本调试例子中，有一个启动调试运行的快捷方式，即单击Navigate下的Debug图标（见图1-13）。不过要小心，每当启动一个新调试运行时，都需要单击一个红色的Terminate方框来退出正在运行的那个调试。一个这样的方框在Debug视图的工具栏中，另一个在控制台视图中。Debug视图还有一个双X图标，Remove All Terminated Launches。

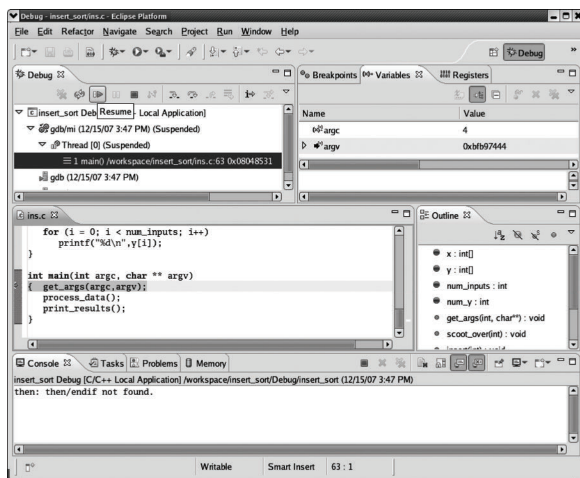


图1-13　调试运行的开始

图1-13显示了启动调试之后出现的屏幕。虽然人们可以在Eclipse调试对话框中设置启动行，但是通常默认将一个自动断点放在第一行可执行代码上。在图1-13中，可以从下面的代码行的左

边空中的断点符号上看到这一点：

```
{ get_args(argc,argv);
```

这一行也是突出显示的，因为它是要执行的代码行。通过在Debug视图工具栏中单击Resume图标来前进并执行，这个工具栏在窗口中弹出的一个框上面，因为你将鼠标指针移到了该图标处。

在示例GDB会话中，程序的第一个版本有一个无限循环，程序被挂起。这里当然会看到同样的现象，控制台视图中没有输出。你需要关闭这个程序。然而，你不想通过单击那个红色的Terminate方框来做到这一点，因为这也会关闭底层GDB会话。你要留在GDB中，以便查看你位于代码中的何处（即无限循环的位置）分析变量的值，等等。因此，不要选择Terminate操作，而要选择Suspend，单击Debug视图工具栏中的Resume右边的图标。（在Eclipse文献中，这个按钮有时被称为*Pause*，因为它的符号类似于媒体播放器中的暂停操作。）

当单击Suspend后，屏幕如图1-14所示。从图中可以看到在该操作之前，Eclipse打算执行以下行。

```
for (j = 0; j < num_y; j++)  {
```

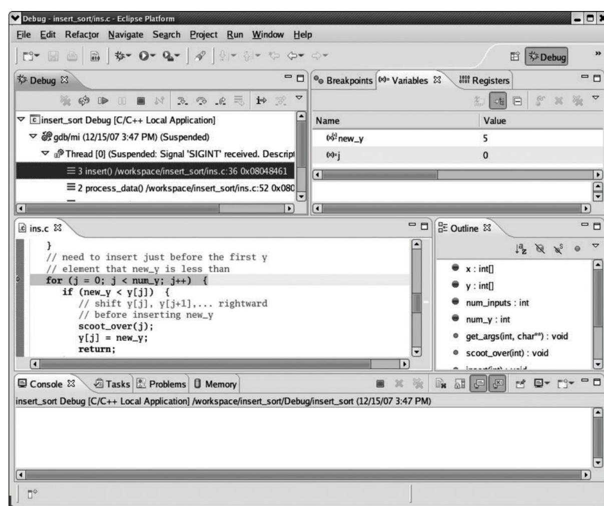


图1-14　挂起的程序

现在可以通过将鼠标指针移到源代码窗口中该变量的任何实例上来分析`num_y`的值（你将发现该值为0），等等。

再次回顾我们前面的GDB会话。当修复了几个程序错误后，程序出现了一个段错误。图1-15显示了那一刻的Eclipse屏幕。

发生的事情是当时我们单击了Resume按钮，因此程序继续运行，突然因为段错误而在如下这行代码处停止：

```
y[k] = y[k-1];
```

奇怪的是，从图1-15中还可以看出，Eclipse没有在Problems选项卡中指出这个问题，而是在Debug选项卡中显示如下错误消息。

```
(Suspended'SIGSEGV' received. Description: Segmentation fault.)
```

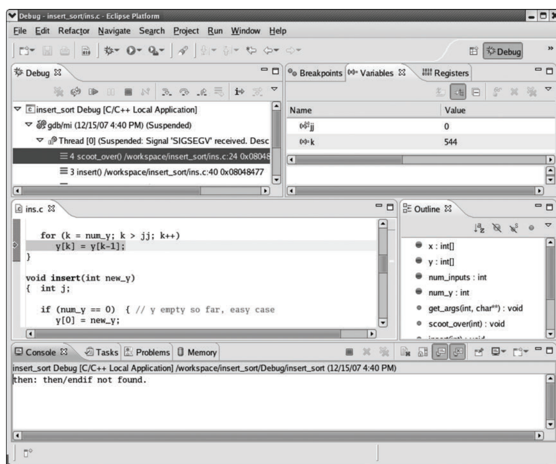


图1-15　段错误

在这个选项卡中可以看到错误发生在从`insert()`中调用的函数`scoot_over()`中。像在GDB示例中那样，你同样可以查询变量的值，比如发现`k=544`超出了范围。

在GDB示例中还设置了条件断点。在Eclipse中通过在所需行的左边空中双击来设置断点。要使该断点为条件断点，那么右击那一行的断点符号，并选择Breakpoint Properties...→New→Common，然后在对话框中填写条件。该对话框如图1-16所示。

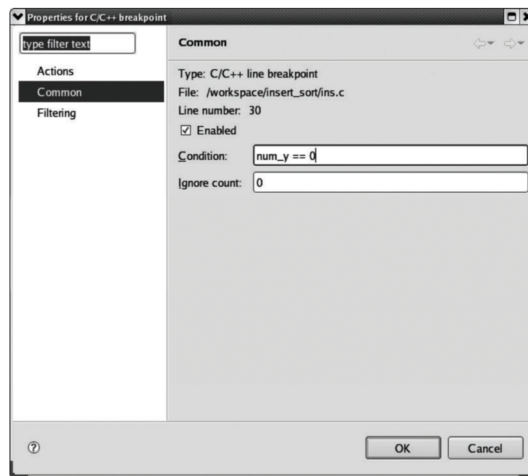


图1-16　使断点成为条件断点

我们还介绍过，在GDB会话中，偶尔会在GDB外面（在单独的终端窗口中）执行程序。在Eclipse中通过选择Run→Run也可以轻松地做到这一点。与惯常一样，结果将显示在控制台视图中。

1.8　启动文件的使用

正如前面提到的，在重新编译代码时，最好不要退出GDB。这样，你的断点和建立的其他各种动作都会保留（比如第3章将介绍的`display`命令）。要是退出GDB，就不得不再次重复键入所有这些内容。

然而，在完成调试前可能需要退出GDB。如果你要离开一段时间甚至离开一天，而且不能保持登录在计算机中，则需要退出GDB。为了不丢失它们，可以将断点和设置的其他命令放在一个GDB启动文件中，然后每次启动GDB时会自动加载它们。

GDB的启动文件默认名为*.gdbinit*。可以将一个文件放在主目录中用于一般用途，另一个文件放在特定项目专用的目录中。例如，将设置断点的命令放在后一个目录的启动文件中。在主目录的*.gdbinit*文件中，还可以存储你开发的一些通用的宏（第2章将会介绍）。

GDB在加载可执行文件之前会读取主目录中的启动文件。因此，要是在主目录的*.gdbinit*文件中有一个命令，比如：

```
break g
```

表示要在函数`g()`上中断，那么GDB总是会在启动时抱怨它不知道该函数。然而，在本地项目目录的启动文件中可以有这行代码，因为本地启动文件是在加载了可执行文件（及其符号表）之后读取的。注意，GDB的这个特性暗示了最好不要将编程项目放在主目录中，因为不能将项目特有的信息放在*.gdbinit*中。

在调用GDB时可以指定启动文件。例如，

```
$ gdb -command=z x
```

表示要在可执行文件*x*上运行GDB，首先要从文件*z*中读取命令。此外，因为DDD只是GDB的一个前端，所以调用DDD也会调用GDB的启动文件。

最后，可以通过选择Edit→Preferences来以各种各样的方式定制DDD。对于Eclipse，选择Windows→Preferences。

第 2 章

停下来环顾程序

像GDB这样的符号调试器当然可以像你那样地运行程序。然而，在可执行文件中神奇般地加入调试符号后，调试器呈现了可以逐行执行源代码的错觉，而不是基于逐个被编译的机器码指令执行。这一看似简单的事实恰好使得符号调试器在调试程序中非常有用。

如果调试器能做的仅仅是运行程序，那么对我们来说没有太大的用处。我们当然也能做同样的事情，而且还更有效率。调试器的好处在于：可以通知它暂停程序的执行。暂停以后，调试器让我们得以检查变量、跟踪执行路径等。

2.1　暂停机制

有3种方式可以通知GDB暂停程序的执行。

- 断点：通知GDB在程序中的特定位置暂停执行。
- 监视点：通知GDB当特定内存位置（或者涉及一个或多个位置的表达式）的值发生变化时暂停执行。
- 捕获点：通知GDB当特定事件发生时暂停执行。

容易混淆的是（一开始），在GDB文档中将这3个机制都称为断点。这可能是因为它们的很多属性和命令都相同。例如，在帮助文档中可以看到，GDB关于删除断点的`delete`命令：

```
(gdb) help delete
Delete some breakpoints or auto-display expressions.
Arguments are breakpoint numbers with spaces in between.
To delete all breakpoints, give no argument.
```

然而，有经验的GDB用户知道帮助文档在这里的真正意思是`delete`命令可以删除所有断点、监视点和捕获点！

2.2　断点概述

断点就像程序中的绊网：在程序中的特定“位置”设置断点，当到达那一点时，调试器会暂停程序的执行（在GDB这样的基于文本的调试器的情况下，会出现命令行提示符）。

GDB中关于“位置”的含义是非常灵活的，它可以指各种源代码行、代码地址、源代码文件

中的行号或者函数的入口等。

下面是一个调试会话片段，说明当GDB在一行代码上中断时发生的事情。在这段代码中，列出了部分源代码，在程序的第35行放置了一个断点，然后运行程序。GDB到达该断点并暂停。

```
(gdb) list
30
31              /* Get the size of file in bytes */
32              if ((fd = open(c.filename, O_RDONLY)) == -1)
33                      (void) die(1, "Can't open file.");
34              (void) stat(c.filename, &fstat);
35              c.filesize = fstat.st_size;
36
(gdb) break 35
Breakpoint 1 at 0x8048ff3: file bed.c, line 35.
(gdb) run
Starting program: binary_editor/bed
Breakpoint 1, main (argc=1, argv=0xbfa3e1f4) at bed.c:35
35              c.filesize = fstat.st_size;
(gdb)
```

让我们解释一下这里发生了什么事：GDB从第30行执行到第34行，但是第35行还没有执行。这可能有点让人迷惑，因为很多人以为GDB显示的是最后执行的代码行，而事实上，它显示的是将要执行的代码行。在本例中，GDB告诉我们第35行是将要执行的下一行源代码。当GDB的执行到达第35行的断点时，可以认为GDB在源代码的第34行和第35行之间等待。

然而你知道，GDB的工作针对的是机器语言指令，而不是源代码行，一行代码可能对应于数行机器语言。GDB之所以可以使用源代码行，是因为可执行文件中包括了额外的信息。虽然这个事实暂时似乎还不太重要，但是在本章讨论单步调试程序时会有一定的涉及。

2.3　跟踪断点

程序员创建的每个断点（包括断点、监视点和捕获点）都被标识为从1开始的唯一整数标识符。这个标识符用来执行该断点上的各种操作。调试器还包括一种列出所有断点及其属性的方式。

2.3.1　GDB中的断点列表

当创建断点时，GBD会告知你分配给该断点的编号。例如，本例中设置的断点：

```
(gdb) break main
Breakpoint 2 at 0x8048824: file efh.c, line 16.
```

被分配的编号是2。如果忘记了分配给哪个断点的编号是什么，可以使用`info breakpoints`命令来提示。

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
```

```
1    breakpoint     keep y   0x08048846 in Initialize_Game at efh.c:26
2    breakpoint     keep y   0x08048824 in main at efh.c:16
        breakpoint already hit 1 time
3    hw watchpoint  keep y              efh.level
4    catch fork     keep y
```

我们将看到这些标识符用来执行断点上的各种各样的操作。为了具体说明，来看一个简单的示例。

上一节中介绍了**delete**命令。通过使用**delete**命令以及断点标识符，可以删除断点1、监视点3及捕获点4。

```
(gdb) delete 1 3 4
```

下面几节将介绍断点标识符的其他用途。

2.3.2 DDD中的断点列表

DDD用户主要使用点选式界面执行断点管理操作，因此断点标识符对于DDD用户不如对GDB用户那么重要。选择Source→Breakpoints会弹出Breakpoints and Watchpoints窗口，其中列出了所有断点，如图2-1所示。

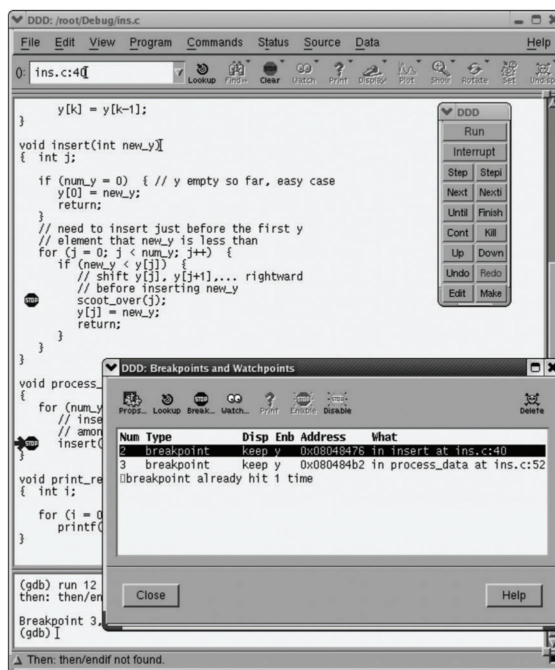


图2-1 在DDD中查看断点

然而前面介绍过，DDD允许使用GBD的基于命令的界面及其提供的GUI。在有些情况下，

GDB提供了不能通过DDD GUI使用的断点操作，但是DDD用户能够通过DDD控制台访问那些特殊的GDB操作。在这些情况下，断点标识符对DDD用户仍然是很有用的。

注意，如果愿意，可以让这个Breakpoints and Watchpoints窗口始终打开，只要将它拖放到屏幕的方便部分即可。

2.3.3　Eclipse中的断点列表

Debug透视图中包括Breakpoints视图。例如，在图2-2中可以看到文件*ins.c*的第30行和第52行目前有两个断点。

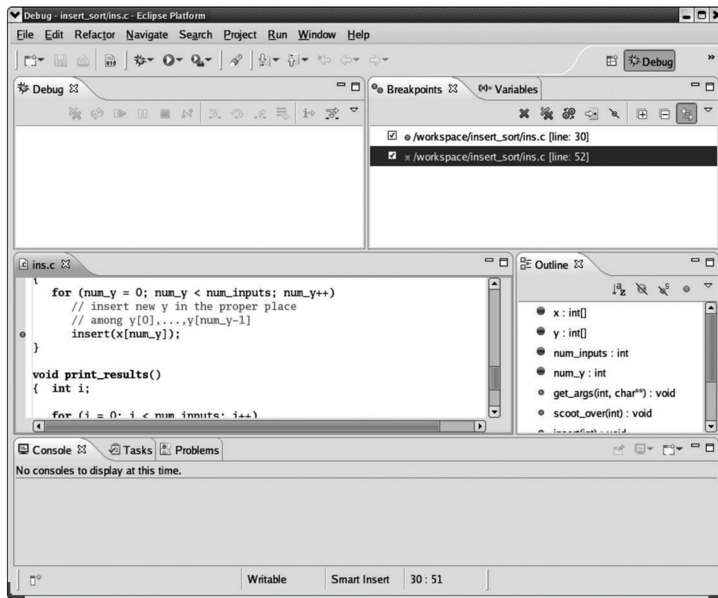


图2-2　在Eclipse中查看断点

可以右击任意断点的入口来检查或修改其属性。另外，双击入口将导致源文件窗口的焦点转移到该断点上。

2.4　设置断点

调试工具通常提供了多种设置断点的机制。

2.4.1　在GDB中设置断点

前面已经介绍过，GDB给人一种逐行运行源代码的错觉。使用GDB时，需要指示在何处暂停执行，以便使用GDB的命令行提示符执行调试活动。本节将介绍如何设置断点，并通知GDB在源代码的何处停止。

GDB中有许多指定断点的方式，下面是一些最常见的方法。

- break *function*

在函数function()的入口（第一行可执行代码）处设置断点。在2.3.1节中提供过这种示例，下述命令在main()的入口处设置断点：

```
(gdb) break main
```

- break *line_number*

在当前活动源代码文件的*line_number*处设置断点。对于多个文件的程序，这要么是上次使用list命令查看其内容的文件，要么是包含*main*()的文件。2.2节中提供过这种情况的示例。

```
(gdb) break 35
```

它在文件*bed.c*中的第35行处设置了一个断点。

- break *filename:line_number*

在源代码文件*filename*的*line_number*处设置断点。如果*filename*不在当前工作目录中，则可以给出相对路径名或者完全路径名来帮助GDB查找该文件，例如：

```
(gdb) break source/bed.c:35
```

- break *filename:function*

在文件*filename*中的函数*function()*的入口处设置断点。重载函数或者使用同名静态函数的程序可能需要使用这种形式，例如：

```
(gdb) break bed.c:parseArguments
```

我们后面将看到，当设置一个断点时，该断点的有效性会持续到删除、禁用或退出GDB时。然而，临时断点是首次到达后就会被自动删除的断点。临时断点使用tbreak命令设置，它与break采用相同类型的参数。例如，tbreak foo.c:10在文件*foo.c*的第10行处设置临时断点。

其他break参数

break命令还有一些别的参数，也许很少用到。

- 使用break +*offset*或break -*offset*，可以在当前选中栈帧中正在执行的源代码行之前或之后*offset*行设置断点。
- break **address*这种形式可用来在虚拟内存地址处设置断点。这对于程序没有调试信息的部分（比如找不到源代码时，或者对于共享库）是必需的。

对于同名函数要加上注释。C++允许重载函数（使用相同的名称定义函数）。甚至在C语言中也可以重载函数，只要使用static限定符声明带文件作用域的函数。使用break *function*会在所有具有相同名称的函数上设置断点。如果要在函数的某个特定实例上设置断点，则需要没有歧义，比如在break命令中给出源代码文件中的行号。

GDB实际设置断点的位置可能与你请求将断点放置的位置不同。不熟悉GDB的开发人员

可能对此感到不安，因此让我们看一下这种奇怪现象的简短示例。来看下面这个短程序，test1.c：

```
1  int main(void)
2  {
3     int i;
4     i = 3;
5
6     return 0;
7  }
```

不加优化地编译这个程序，并尝试在main()的入口处设置断点。你以为断点会放在函数的上方——在第1行、第2行或第3行。虽然人们常常猜测断点会在这些位置，但是他们错了。断点实际上在第4行。

```
$ gcc -g3 -Wall -Wextra -o test1 test1.c
$ gdb test1
(gdb) break main
Breakpoint 1 at 0x6: file test1.c, line 4.
```

第4行不能算是main()的开头行，那么发生了什么事呢？你可能猜到，一个原因是这一行是可执行代码。我们介绍过，GDB实际上是使用机器语言指令工作的，但是有了增强的符号表的魔力后，GDB表现出了使用源代码行的错觉。一般来说这个事实不太重要，但是在这种情况下这一事实却变得很重要。实际上，声明i确实会生成机器码，但是GDB认为这种机器码对于我们的调试目的来说没有用处。[①]因此，当要求GDB在main()的开头中断时，它就在第4行设置了一个断点。

其他断点类型

此外，还有一些break风格的命令，本书不打算深入介绍，因为它们的专用性很强。

hbreak命令设置硬件辅助断点。这是可以在内存中设置的断点，不需要修改该内存位置的内容。这需要硬件支持，主要用于EEPROM/ROM 调试。此外，还有一个设置临时硬件辅助断点的thbreak命令。hbreak和thbreak命令采用与break和tbreak相同类型的参数。

rbreak命令采用grep风格的正则表达式，并在匹配该正则表达式的任何函数的入口处设置断点。关于rbreak要知道两件事。首先，请记住rbreak使用grep风格的正则表达式，而不是Perl风格的正则表达式或者shell风格的文件名替换（globbing）。举例来说，这意味着rbreak func*会将断点放在名为func和funcc()的函数上，而不会放在function()上。其次，在rbreak的参数前面或后面有一个隐含的.*，因此，如果不希望rbreak func在afunc()上设置断点，应当使用rbreak ^func。

① 可以通过-S选项查看GCC生成的机器码。

当打开优化来编译程序时，这个问题可能变得更糟糕。让我们来看一下。打开优化，重新编译该程序。

```
$ gcc -O9 -g3 -Wall -Wextra -o test1 test1.c
$ gdb test1
(gdb) break main
Breakpoint 1 at 0x3: file test1.c, line 6.
```


我们要求在`main()`的开头放置一个断点，但是GDB在`main()`的最后一行放置了一个断点。到底发生了什么事？答案与前面相同，只是GCC扮演了一个更主动的角色。在优化打开的情况下，GCC注意到虽然为`i`赋予了一个值，但是永远用不到这个值。因此，为努力生成更有效的代码，GCC就把第3行和第4行代码优化掉了。GCC永远不会生成这几行的机器指令。因此，生成机器指令的第一行源代码恰好是`main()`的最后一行。这是在调试完成前不应当优化代码的原因之一。[①]

捕　获　点

C++程序员或许会想了解设置捕获点的`catch`命令。捕获点类似于断点，但是能够通过如抛出异常、捕获异常、发信号通知、调用`fork()`、加载和卸载库等很多事件触发。

本书将详细介绍断点和监视点，而让读者参阅GDB文档了解关于捕获点的更多信息。

知道所有这些情况后，如果发现设置断点后没有正好在你预期的地方产生断点，你现在就知道是为什么了，不用大惊小怪。

另外我们还要提一句，当同一行源代码上有多个断点时会发生什么情况。当GDB使用多个断点中断一行源代码时，它只会中断一次。换言之，当它到达该行代码时，如果恢复执行，会忽略恰好在同一行上的其他断点。事实上，GDB知道是哪个断点“触发”了程序停止执行。在具有多个断点的代码行上，触发中断的断点将是标识符编号最小的断点。

2.4.2　在DDD中设置断点

为了使用DDD设置断点，在源窗口中查找要设置断点的代码行。将光标放在那一行上的任意空白处，右击，这时会弹出一个菜单。将鼠标向下拖动，直到突出显示Set Breakpoint选项，然后释放鼠标按键。这时应当在设置断点的代码行旁边看到一个红色停止记号。如果没有使用断点做任何特别的事，比如使它成为条件断点（将在2.10节讨论），那么一种快捷方式是直接双击这行代码。

如果一直在用DDD，你可能会注意到，当在一行代码旁边按下鼠标右键时，弹出菜单中会包含选项Set Temporary Breakpoint。这就是在DDD中设置临时断点（当首次到达后就消失的断点）

---

① 实际上，在打开优化编译可执行文件时，有些调试器真的会阻塞。GDB是可以调试优化代码的为数不多的调试器之一，但是正如你看到的，结果会有问题。

的方法，这种方法调用了GDB的`tbreak`命令。

同样，不要忘记DDD实际上是GDB的前端。可以在DDD中使用DDD的控制台窗口执行任意GDB风格的中断命令。有时需要这种办法，如果程序非常大或者是多文件程序，那么使用GDB语法设置断点会很方便。事实上，之所以有时有必要这样做，是因为并非GDB的所有断点命令都可以从DDD界面中调用。

2.4.3　在Eclipse中设置断点

为了在Eclipse中对给定代码行设置断点，双击该行代码。这时会出现一个断点符号，如图2-2中的如下代码行所示。

```
insert(x[num_y]);
```

为了设置临时断点，单击该行代码，然后在源代码窗口中右击，并选择Run to Line。然而要注意，只有当目标行与当前位置位于同一个函数中时，Run to Line操作才会起作用，而且要在重新输入和遇到这一行前没有退出该函数才行。

2.5　展开 GDB 示例

因为有很多信息，所以有必要举一个可以照着做的设置断点的简短例子。来看下面的多文件C代码。

main.c:

```
#include <stdio.h>
void swap(int *a, int *b);

int main(void)
{
   int i = 3;
   int j = 5;

   printf("i: %d, j: %d\n", i, j);
   swap(&i, &j);
   printf("i: %d, j: %d\n", i, j);

   return 0;
}
```

swapper.c:

```
void swap(int *a, int *b)
{
   int c = *a;
   *a = *b;
   *b = c;
}
```

编译该代码并在可执行文件上运行GDB。

```
$ gcc -g3 -Wall -Wextra -c main.c swapper.c
$ gcc -o swap main.o swapper.o
$ gdb swap
```

说明　本例是本书中首次编译多文件C程序，因此这里作一下说明。上述编译过程的第一行产生了两个目标文件，其中包含带调试信息的未决目标代码。第二行将目标文件连接到一个包含所有调试信息的可执行文件中。在连接过程中不需要使用GCC的-g开关选项。

启动调试会话时在main()中设置断点是很常见的。这一操作在该函数的第一行上设置断点。[①]

```
(gdb) break main
Breakpoint 1 at 0x80483f6: file main.c, line 6.
```


下面的所有示例都在函数swap()的第一行上设置了断点。虽然看上去可能不同，但是它们做的都是同一件事：在swap()的上方中断。

```
(gdb) break swapper.c:1
Breakpoint 2 at 0x8048454: file swapper.c, line 1.
(gdb) break swapper.c:swap
Breakpoint 3 at 0x804845a: file swapper.c, line 3.
(gdb) break swap
Note: breakpoint 3 also set at pc 0x804845a.
Breakpoint 4 at 0x804845a: file swapper.c, line 3.
```

在任何给定时间，GDB都有一个焦点，可以将它看作当前“活动”文件。这意味着除非对命令做了限定，否则都是在具有GDB的焦点的文件上执行命令。默认情况下，具有GDB的初始焦点的文件是包含main()函数的文件，但是当发生如下任一动作时，焦点会转移到不同的文件上。

- 向不同的源文件应用list命令。
- 进入位于不同的源代码文件中的代码。
- 当在不同的源代码文件中执行代码时GDB遇到断点。

让我们看一个示例。虽然断点是在*swapper.c*中设置的，但是我们实际上没有列出该文件中的代码。因此，焦点仍然位于*main.c*上。可以通过在第6行上设置断点来验证这一点。当不带文件名地设置这个断点时，GDB会在当前活动文件的第6行上设置断点。

```
(gdb) break 6
Breakpoint 5 at 0x8048404: file main.c, line 6.
```

可以肯定的是，*main.c*具有焦点：当启动GDB时，仅仅通过行号设置的断点是在包含main()的文件中设置的。可以通过列出*swapper.c*中的代码来改变焦点。

```
(gdb) list swap
1   void swap(int *a, int *b)
```

① 断点可能不是恰好在main()的第一行上，但是会在靠近第一行的位置。然而，与我们前面介绍这一点的示例不同，这里的代码行是可执行的，因为它为i赋了值。

```
2  {
3     int c = *a;
4     *a = *b;
5     *b = c;
6  }
```

可以试着在第6行上设置另一个断点，来确认*swapper.c*现在具有焦点。

```
(gdb) break 6
Breakpoint 6 at 0x8048474: file swapper.c, line 6.
```

可以看出，断点确实是在*swapper.c*的第6行上设置的。然后将在*swapper.c*的第4行上设置一个临时断点：

```
(gdb) tbreak swapper.c:4
Breakpoint 7 at 0x8048462: file swapper.c, line 4.
```

最后，使用2.3.1节介绍的`info breakpoints`命令，来展示一下刚才设置的所有断点的阵容。

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483f6 in main at main.c:6
2   breakpoint     keep y   0x08048454 in swap at swapper.c:1
3   breakpoint     keep y   0x0804845a in swap at swapper.c:3
4   breakpoint     keep y   0x0804845a in swap at swapper.c:3
5   breakpoint     keep y   0x08048404 in main at main.c:9
6   breakpoint     keep y   0x08048474 in swap at swapper.c:6
7   breakpoint     del  y   0x08048462 in swap at swapper.c:4
```

到后面结束GDB会话时，再使用`quit`命令离开GDB。

```
(gdb) quit
$
```

2.6　断点的持久性

我们上面说的“后面”是指在调试会话期间不应退出GDB。例如，当发现并修复了一个程序错误，但是其他程序错误仍然存在时，不应当退出GDB然后重新进入来使用程序的新版本。这样做有些不必要地繁琐，而且还会不得不重新进入断点。

如果在修改和重新编译代码时没有退出GDB，那么在下次执行GDB的`run`命令时，GDB会感知到代码已修改，并自动重新加载新版本。

然而要注意，断点是会“移动”的。例如，来看下面这个简单程序。

```
1  main()
2  {  int x,y;
```

```
3      x = 1;
4      y = 2;
5   }
```

编译该程序，进入GDB，并在第4行设置一个断点。

```
(gdb) l
1       main()
2       {  int x,y;
3          x = 1;
4          y = 2;
5       }
(gdb) b 4
Breakpoint 1 at 0x804830b: file a.c, line 4.
(gdb) r
Starting program: /usr/home/matloff/Tmp/tmp1/a.out

Breakpoint 1, main () at a.c:4
4          y = 2;
```

一切正常。但是假设现在添加一行源代码。

```
1   main()
2   {  int x,y;
3      x = 1;
4      x++;
5      y = 2;
6   }
```

然后重新编译（同样，记住并没有离开GDB），并再次执行GDB的run命令。

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
`/usr/home/matloff/Tmp/tmp1/a.out' has changed; re-reading symbols.

Starting program: /usr/home/matloff/Tmp/tmp1/a.out

Breakpoint 1, main () at a.c:4
4          x++;
```

GDB确实重新加载了新代码，但是断点似乎从下面的语句：

```
y = 2;
```

移到了如下语句：

```
x++;
```

仔细看一下将发现断点实际上根本没有移动：断点本来在第4行，现在仍然在第4行。但是那一行不再是原来在其上设置断点的语句。因此，需要通过删除这个断点并设置一个新断点来移动断点。（DDD中的做法容易得多，参见2.7.5节。）

最后，假设因为该吃饭、睡觉或休息，而需要结束当前调试会话。如果你一般没有持续开机的习惯，就需要退出调试器。有没有什么方法可以保存断点呢？

对于GDB和DDD，在某种程度上答案是肯定的。可以将断点放在源代码所在目录（或者从中调用GDB的目录）的*.gdbinit*启动文件中。

如果在使用Eclipse，则比较幸运，因为所有断点都会被自动保存，并且在下一个Eclipse会话中恢复。

2.7　删除和禁用断点

在调试会话期间，有时会发现有的断点不再有用。如果确认不再需要该断点，可以删除它。

也可能是这样的情况：你认为断点会在调试会话之后还有用，也许你不想删除断点，而是打算将它虚置起来，让调试器暂时不会在该断点处中断。这称为禁用断点。如果以后再次需要，可以重新启用断点。

本节介绍删除和禁用断点，这里提到的一切方法都同样适用于监视点。

2.7.1　在GDB中删除断点

如果确认不再需要当前断点（也许是因为修复了那个特定程序错误），那么可以删除该断点。GDB中有两个用来删除断点的命令。`delete`命令用来基于标识符删除断点，`clear`命令使用与2.4.1节所介绍的创建断点相同的语法删除断点。

- `delete` *`breakpoint_list`*

删除断点使用数值标识符（已经在2.3节介绍）。断点可以是一个数字，比如`delete 2`删除第二个断点；也可以是一系列数字，比如`delete 2 4`删除第二个和第四个断点。

- `delete`

删除所有断点。除非执行`set confirm off`命令（它也可以放在*.gdbinit*启动文件中），否则GDB会要求确认删除操作。

- `clear`

清除GDB将执行的下一个指令处的断点。这种方法适用于要删除GDB已经到达的断点的情况。

- `clear` *`function`*、`clear` *`filename:function`*、`clear` *`line_number`*和`clear` *`filename:line_number`*

这些命令根据位置清除断点，工作方式与对应的`break`命令相似。

例如，假设使用如下命令在`foo()`的入口处设置断点。

```
(gdb) break foo
Breakpoint 2 at 0x804843a: file test.c, line 22.
```

可以用来删除该断点的代码为：

```
(gdb) clear foo
Deleted breakpoint 2
```

或者

```
(gdb) delete 2
Deleted breakpoint 2
```


2.7.2　在GDB中禁用断点

每个断点都可以被启用或禁用。只有当GDB遇到启用的断点时，才会暂停程序的执行；它会忽略禁用的断点。默认情况下，断点的生命期从启用时开始。

为什么要禁用断点呢？在调试会话期间，会遇到大量断点。对于经常重复的循环结构或函数，这种情况使得调试极不方便。如果要保留断点以便以后使用，同时又不希望GDB停止执行，可以禁用它们，在以后需要时再启用。

使用`disable` *`breakpoint-list`*命令禁用断点，使用`enable` *`breakpoint-list`*命令启用断点，其中*`breakpoint-list`*是由空格分隔开的多个断点标识符。例如，

```
(gdb) disable 3
```

将禁用第三个断点。类似地，

```
(gdb) enable 1 5
```

将启用第一个和第五个断点。

不带任何参数地执行`disable`命令将禁用所有现存断点。类似地，不带参数地执行`enable`命令会启用所有现有断点。

还有一个`enable once`命令，在该断点下次引起GDB暂停执行后禁用它。语法为：

```
enable once breakpoint-list
```

例如，`enable once 3`会使得断点3在下次导致GDB停止程序的执行后被禁用。这个命令与`tbreak`命令非常类似，但是当遇到断点时，它是禁用断点，而不是删除断点。

2.7.3　在DDD中删除和禁用断点

在DDD中删除断点与设置断点一样方便。与设置断点时一样，将光标放在红色停止符号上，并右击鼠标。弹出菜单中将有一个选项是Delete Breakpoint。按住鼠标右键，并拖动鼠标，直到突出显示这个位置。然后释放鼠标按键，将看到红色停止符号消失了，表示断点已被删除。

在DDD中禁用断点与删除断点非常相似。右击并按住红色停止符号，并选择Disable Breakpoint。红色停止符号会变成灰色，表示断点仍然存在，但已经禁用了。

还可以使用DDD的Breakpoints and Watchpoints窗口，如图2-1所示。可以单击断点入口来突出显示，然后选择Delete、Disable或Enable。

事实上，可以通过将鼠标拖到上面来突出显示该窗口中的几个条目，如图2-3所示。因此可以立即删除、禁用或启用多个断点。

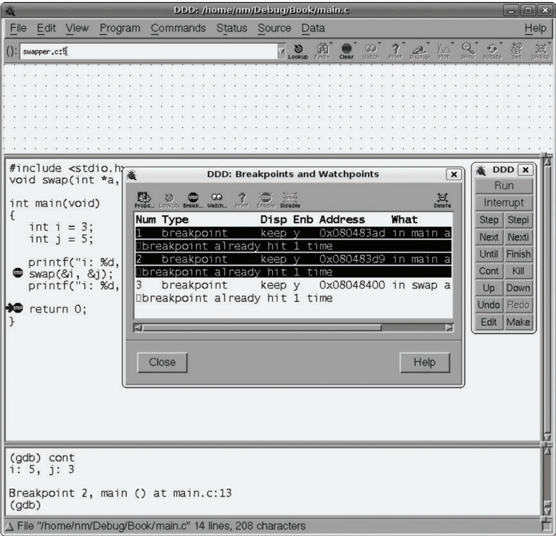


图2-3　在DDD中删除/禁用/启用多个断点

当在断点窗口中单击Delete按钮后，屏幕看上去如图2-4所示。显然，有两个老断点现在消失了。

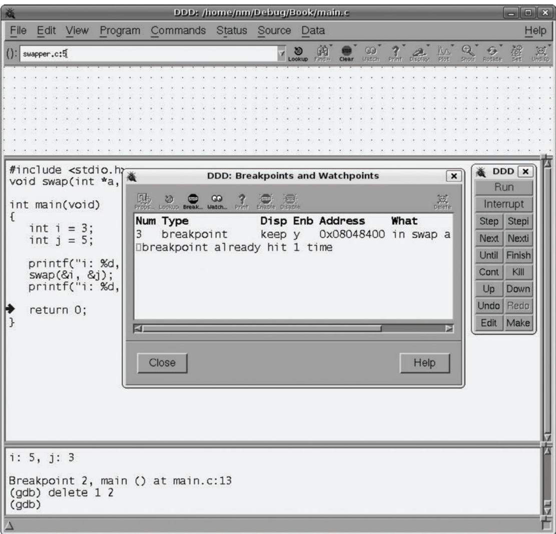


图2-4　两个被删除的断点

2.7.4　在Eclipse中删除和禁用断点

与在DDD中一样，可以在Eclipse中通过单击当前活动的代码行中的断点符号来删除或禁用断点。这时会弹出一个菜单，如图2-5所示。注意，Toggle选项意味着删除断点，而Disable/Enable的意思显而易见。

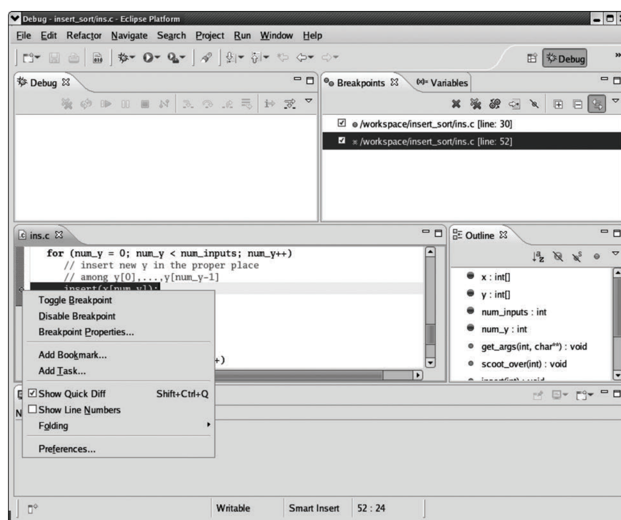


图2-5　在Eclipse中删除/禁用断点

类似于DDD的断点窗口，Eclipse有Breakpoint视图，在图2-5所示界面的右上方。与DDD中需要请求Breakpoints and Watchpoints窗口的情况不同，Eclipse的Breakpoints视图是在Debug透视图中自动显示的。（不过，如果屏幕空间不够，可以单击右上角的X将它隐藏起来。如果以后要使它恢复，选择Window→Show Views→Breakpoints。）

Eclipse Breakpoints视图的优点是可以单击双X图标（Remove All Breakpoints）。在编程过程中需要这么做的机会往往比你想象的要多。在长调试会话中的有些地方，会发现你以前设置的断点现在没有一个是有用的，因此要把它们全部删除。

2.7.5　在DDD中“移动”断点

DDD的另一个真正优秀的功能是拖放断点。单击并按住源窗口中的断点停止符号，只要保持按下鼠标左键，就可以将断点拖到源代码中的另一个位置。“幕后”发生的事情是DDD删除了原来的断点，并设置了一个具有相同属性的新断点。因此，你将发现新断点与老断点完全相同，只是数字标识符不同。当然，也可以使用GDB做这件事，但是DDD加速了这一过程。

图2-6说明了这一点，这是在断点移动操作中的截图。下面的代码行上原来有一个断点。

```
if (new_y < y[i]) {
```

我们希望将该断点移到如下这行代码上。

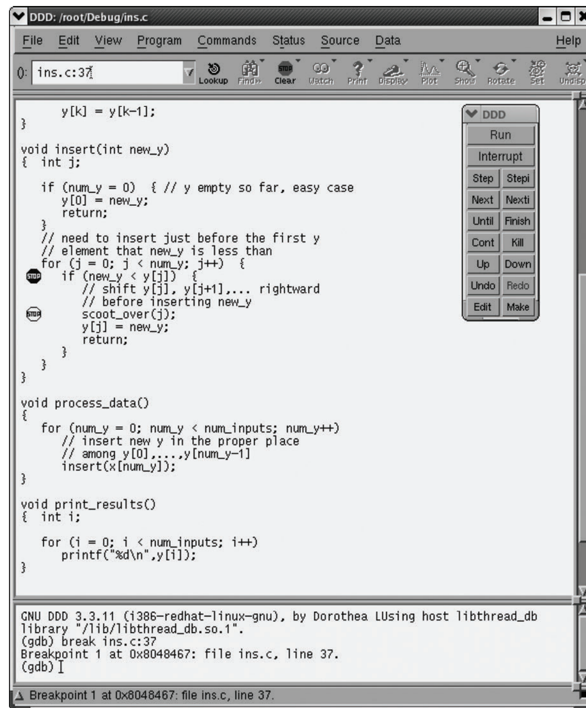


图2-6　在DDD中“移动”断点

```
scoot_over(j);
```

我们单击老代码行上的停止符号，并将它拖到新代码行上。在该图中，我们还没有释放鼠标按键，因此，原来的停止符号仍然在那儿，新行上出现了一个“空”停止符号。一旦释放鼠标按键，老代码行上的停止符号就会消失，新行上出现一个停止符号。

为什么这样做很有用呢？首先，如果处于要删除一个断点并添加另一个断点的情况下，这样可以一气呵成完成两个操作。但是更重要的是，如2.6节所提到的那样，当添加或删除源代码行时，有部分行的行号发生了变化，从而将现有断点“转移到”了不打算设置断点的行上。在GDB中，这就需要在每个受影响的断点处进行删除断点和创建新断点的操作。这样做是很繁琐的，尤其是当部分断点有附加条件时更麻烦。但是在DDD中，可以直接将停止符号图标拖到新位置，这样操作非常方便。

2.7.6　DDD中的Undo/Redo断点动作

DDD的一个真正优秀的功能是Undo/Redo，可通过单击Edit用到。以图2-3和图2-4中的情况为例。假设你突然意识到不想删除那两个断点（也许只是希望禁用它们），可以选择Edit→Undo Delete，如图2-7所示。（也可以单击命令工具中的Undo，但是通过Edit访问会更好，这样DDD会提醒我们将撤销什么操作。）

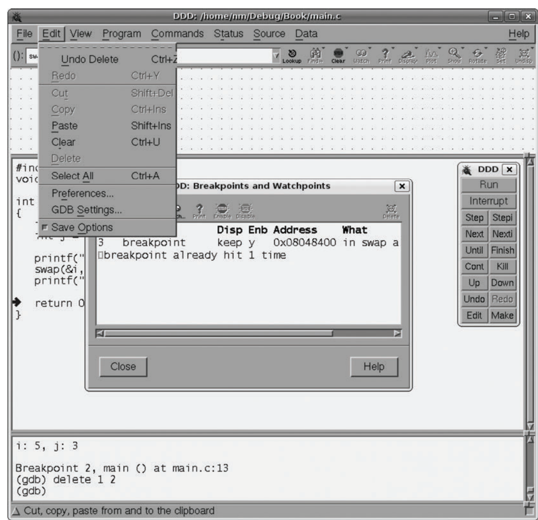


图2-7　在DDD中恢复两个断点

2.8　进一步介绍浏览断点属性

每个断点都有各种属性——行号、加在断点上的条件（如果有的话）、当前启用/禁用状态等。我们在2.3节中介绍了一点儿关于跟踪这些属性的内容，现在我们将详细介绍。

2.8.1　GDB

2.3.1节中介绍过，创建的每个断点都被赋予了唯一的整数标识符。设置的第一个断点被赋予“1”，其后的每个新断点都被赋予上一个标识符加1。每个断点也有一些控制调整其行为的属性。使用唯一标识符，可以分别调整各个断点的属性。

可以使用`info breakpoints`命令（简写为`i b`）来列出设置的所有断点及其属性。`info breakpoints`的输出看上去大体如下：

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048404 in main at int_swap.c:9
        breakpoint already hit 1 time
2   breakpoint     keep n   0x08048447 in main at int_swap.c:14
3   breakpoint     keep y   0x08048460 in swap at int_swap.c:20
        breakpoint already hit 1 time
4   hw watchpoint  keep y              counter
```

让我们详细分析一下`info breakpoints`的这一输出。

(1) 标识符（`Num`）：断点的唯一标识符。

(2) 类型（`Type`）：这个字段指出该断点是断点、监视点还是捕获点。

(3) 部署（Disp）：每个断点都有一个部署，指示断点下次引起GDB暂停程序的执行后该断点上会发生什么事情。可能的部署有以下3种。

- 保持（keep），下次到达断点后不改变断点。这是新建断点的默认部署。
- 删除（del），下次到达断点后删除该断点。使用tbreak命令创建的任何断点都是这样的断点（见2.4.1节）。
- 禁用（dis），下次到达断点时会禁用该断点。这是使用enable once命令设置的断点（见2.7.2节）。

(4) 启用状态（Enb）：这个字段说明断点当前是启用还是禁用的。

(5) 地址（Address）：这是内存中设置断点的位置。它主要供汇编语言程序员或者试图调试没有用扩充的符号表编译的可执行文件的人使用。

(6) 位置（What）：正如我们讨论过的，各个断点位于源代码中的特定行上。What字段显示了断点所在位置的行号和文件名。

对于监视点，这个字段表示正在监视哪个变量。因为变量实际上是带名称的内存地址，而内存地址则是一个位置。

可以看到，除了列出所有断点及其属性，使用i b命令也能指出特定断点引起GDB停止程序执行多少次。例如，如果循环中有一个断点，这个命令马上会告诉你到目前为止循环执行了多少次迭代，这会非常有用。

2.8.2 DDD

从图2-1中可以看出，DDD的Breakpoints and Watchpoints窗口提供的信息与GDB的info breakpoints命令提供的信息相同。然而，DDD的窗口方式比GDB的命令更方便，因为可以经常性地显示这个窗口（即将它放在屏幕边上），因此不需要在每次要查看断点时都执行命令。

2.8.3 Eclipse

再次，如前面图2-2所示，Eclipse的Breakpoints视图可以经常地显示断点及其属性。Eclipse的信息比DDD的信息略少，因为它没有指出到目前为止已遇到断点多少次（甚至Properties窗口中都没有这一信息）。

2.9 恢复执行

知道如何指示调试器在何处或何时暂停程序的执行很重要，但是知道如何指示调试器恢复执行同样重要。毕竟，仅仅查看变量可能不够。有时需要知道变量的值是如何与代码的其余部分交互的。

我们在第1章介绍过确认原则：要一处处确认某些变量有你认为它们应该具有的值，直至遇到一个与你的预期不符之处。然后这种不符将是一种线索，很可能就是程序错误所在的位置。但是通常这种不符要经过多个断点处暂停并恢复执行时才会发生（或者在同一个断点上多次暂停并恢复）。因此，在断点处恢复执行与设置断点本身同样重要，这就是为什么调试工具通常会有相当丰富的恢复执行的方法。

恢复执行的方法有3类。第一类是使用step和next"单步"调试程序，仅执行代码的下一行然后再次暂停。第二类由使用continue组成，使 GDB无条件地恢复程序的执行，直到它遇到另一个断点或者程序结束。最后一类方法涉及条件：用finish或until命令恢复。在这种情况下，GDB会恢复执行，程序继续运行直到遇到某个预先确定的条件（比如，到达函数的末尾），或者到达另一个断点，或者程序完成。

下面将依次介绍GDB的各种恢复执行的方法，然后介绍如何在DDD和Eclipse中执行这样的操作。

2.9.1 在GDB中

本节首先讨论GDB在断点处暂停后的各种恢复执行的方法。

1. 使用step和next单步调试

一旦GDB在断点处停止，可以使用next（简写为n）和step（简写为s）命令来单步调试代码。当触发了断点，并且GDB暂停后，可以使用next和step来执行紧接着的下一行代码。当执行了这一行后，GDB会再次暂停，并给出一个命令行提示符。让我们看一下这一过程的实际使用，来看代码清单2-1。

代码清单2-1 *swapflaw.c*

```
1  /* swapflaw.c: A flawed function that swaps two integers. */
2  #include <stdio.h>
3  void swap(int a, int b);
4
5  int main(void)
6  {
7     int i = 4;
8     int j = 6;
9
10    printf("i: %d, j: %d\n", i, j);
11    swap(i, j);
12    printf("i: %d, j: %d\n", i, j);
13
14    return 0;
15 }
16
17 void swap(int a, int b)
18 {
19    int c = a;
20    a = b;
21    b = c;
22 }
```

我们将在main()的入口处设置一个断点，并在GDB中运行程序。

```
$ gcc -g3 -Wall -Wextra -o swapflaw swapflaw.c
$ gdb swapflaw
(gdb) break main
Breakpoint 1 at 0x80483f6: file swapflaw.c, line 7.
(gdb) run
Starting program: swapflaw
Breakpoint 1, main () at swapflaw.c:7
7               int i = 4;
```

GDB现在位于程序的第7行，这意味着第7行还没有执行。我们可以使用next命令来执行这行代码，这样来到第8行前面：

```
(gdb) next
8               int j = 6;
```

我们将使用step来执行下一行代码（第8行），执行后使我们移到第10行。

```
(gdb) step
10              printf("i: %d, j: %d\n", i, j);
```

我们看到next和step都能执行下一行代码。因此一个重要问题是：“这两个命令的不同之处在哪里？”它们似乎都能执行下一行代码。这两个命令的区别是它们如何处理函数调用：next执行函数，不会在其中暂停，然后在调用之后的第一条语句处暂停，而step在函数中的第一个语句处暂停。

对swap()的调用出现在第11行。让我们比较一下next和step的效果。

使用step：

```
(gdb) step
i: 4, j: 6
11      swap(i, j);
(gdb) step
swap (a=4, b=6) at swapflaw.c:19
19      int c = a;
(gdb) step
20      a = b;
(gdb) step
21      b = c;
(gdb) step
22      }
(gdb) step
main () at swapflaw.c:12
12      printf("i: %d, j: %d\n", i, j);
(gdb) step
i: 4, j: 6
14      return 0;
```

使用next：

```
(gdb) next
i: 4, j: 6
11      swap(i, j);
(gdb) next
12      printf("i: %d, j: %d\n", i, j);
(gdb) next
i: 4, j: 6
14      return 0;
```

step命令的运行方式与你预期的可能一样。该命令在第10行执行printf()，然后在第11行[①]调用swap()，然后它开始执行swap()中的代码行。这称为单步进入（stepping into）函数。一旦我们用step遍历了swap()的所有代码行，step就会把我们带回到main()。

相反地，似乎next永远不会离开main()。这是两个命令的主要区别。next将函数调用看做一行代码，并在一个操作中执行整个函数，这称为单步越过（stepping over）函数。

然而，不要受假相蒙蔽。虽然看上去似乎next越过了swap()的主体，但是它并未真的单步“越过”任何内容。GDB安静地执行swap()的每一行，不向我们显示任何细节（虽然它显示了swap()可以输出的任何屏幕输出），并且没有提示我们执行函数中的各行代码。

单步进入函数（step命令的操作）和单步越过函数（next命令的操作）之间的区别是相当重要的概念，所以为了不厌其烦地强调其重要性，我们用图来演示next和step之间的区别，图中用箭头显示程序执行顺序。

图2-8说明了step命令的行为。想象一下程序在第一个printf()语句处暂停。该图显示了各个step语句会将我们带到何处。

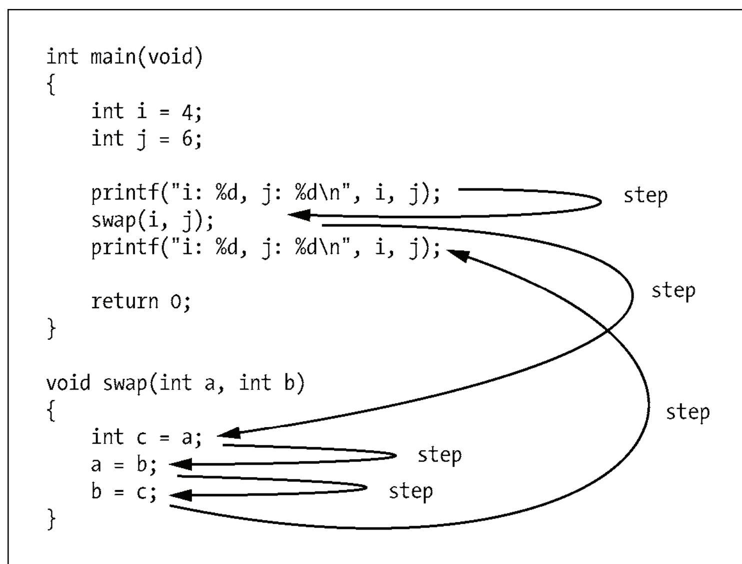


图2-8 step单步进入函数

图2-9说明了同样的事情，不过使用的是next。

应该使用next还是step，取决于要做的事情。如果是在没有函数调用的那部分代码中，那么可以随意。这时两个命令是完全相同的。

然而，如果调试中要单步进入一个你知道没有程序错误（或者与你在试图跟踪的程序错误无

① 你可能会疑惑，为什么step没有将你带到函数printf()的第一行。原因是GDB不会在不具有调试信息的代码（即符号表）内停止。从C库中引入的函数printf()是这样的一个代码示例。

关）的函数中，那么显然使用`next`可以避免单步调试该函数的每一行。

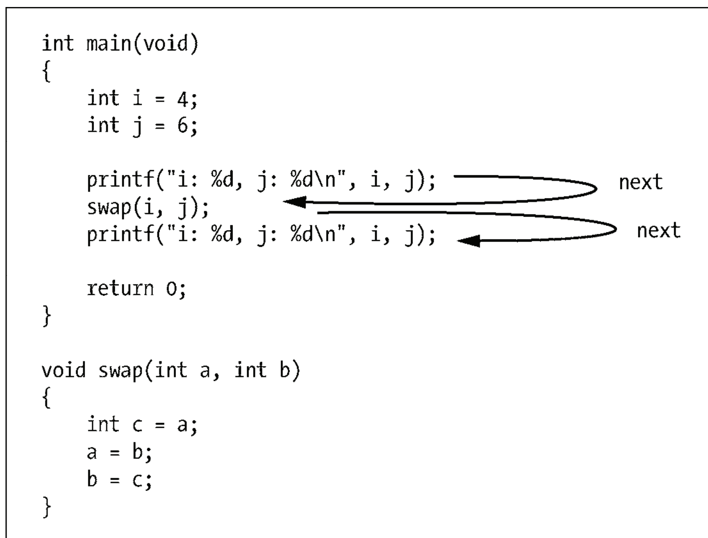


图2-9　next单步越过函数

第1章的一般调试原则之一是采用自顶向下的方法进行调试。如果在单步调试源代码时遇到函数调用，那么一般使用`next`比使用`step`更好。在这种情况下使用`next`之后，立即要检查调用的结果是否正确。如果正确，那么程序错误很可能不在函数中，这意味着使用`next`而不是`step`能够节省单步调试每一行的时间和精力。另一方面，如果函数的结果不正确，可以在函数调用时设置一个临时断点来重新运行程序，然后用`step`进入函数。

`next`和`step`命令都采用一个可选的数值参数，表示要使用`next`或`step`执行的额外行数。换言之，`next 3`与在一行中键入`next` 三次（或者键入`next`一次，然后按下ENTER键两次）的作用相同。[①]图2-10说明了`next 3`的作用。

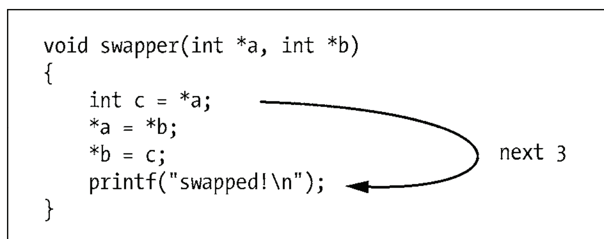


图2-10　带次数的next

① Vim用户应该很习惯指定给定命令次数的含义。

2. 使用continue恢复程序执行

第二种恢复执行的方法是使用continue命令，简写为c。与仅执行一行代码的step和next相反，这个命令使GDB恢复程序的执行，直到触发断点或者程序结束。

continue命令可以接受一个可选的整数参数n。这个数字要求GDB忽略下面n个断点。例如，continue 3让GDB恢复程序执行，并忽略接下来的3个断点。

3. 使用finish恢复程序执行

一旦触发了断点，就使用next和step命令逐行执行程序。有时这是一个痛苦的过程。例如，假设GDB到达了函数中的一个断点。你已查看了几个变量，并且收集了需要的所有信息。这时，你没有兴趣也不需要单步调试函数的其余部分，想返回到这个函数之前GDB所在的调用函数。然而，如果要做的只是跳过函数的其余部分，那么再设置一个无关断点并使用continue似乎比较浪费。这时应该使用finish命令。

finish命令（简写为fin）指示GDB恢复执行，直到恰好在当前栈帧完成之后为止。也就是说，这意味着如果你在一个不是main()的函数中，finish命令会导致GDB恢复执行，直到恰好在函数返回之后为止。图2-11说明了finish的使用。

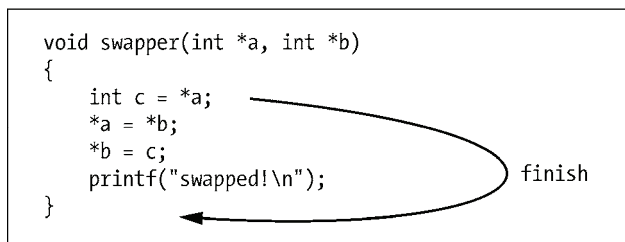


图2-11 finish恢复执行，直到当前函数返回为止

虽然可以键入next 3而不是finish，但是键入后者更容易，这样会对行进行计数（6行以上的内容都显得多余）。

看上去finish似乎没有执行每一行代码，因为它将你带到了函数的底部，但是实际上它执行了每一行代码。除非为了显示程序的输出，否则GDB执行每行代码时不会暂停[①]。

finish的另一个常见用途是当不小心单步进入原本希望单步越过的函数时（换言之，当需要使用next时使用了step）。在这种情况下，使用finish可以将你正好放回到使用next会到的位置。

如果在一个递归函数中，finish只会将你带到递归的上一层。这是因为每次调用都被看做是一次函数调用，因为每次调用都有各自的栈帧。如果要在递归层次较高时完全退出递归函数，那么更适合使用临时断点及continue，或者用until命令。下面我们将讨论until。

4. 使用until恢复程序执行

前面介绍过，finish命令在不进一步在函数中暂停（除了中间断点）的情况下完成当前函数的执行。类似地，until命令（简写为u）通常用来在不进一步在循环中暂停（除了循环中的中间

① 如果有任何介于其间的断点，finish都会在其上暂停。

断点）的情况下完成正在执行的循环。来看如下代码片断。

```
...previous code...

int i = 9999;
while (i--) {
   printf("i is %d\n", i);
   ... lots of code ...
}

...future code...
```

假设GDB在`while`语句的一个断点上停止，你查看了一些变量，现在打算离开循环来调试“未来的代码”。

问题是`i`相当大，使用`next`来完成循环不是办法，而又不能用`finish`，因为该命令正好通过“未来的代码”，并将我们带出函数。在这种情况下可以在未来代码处设置一个临时断点并使用`continue`。这恰好是`until`要来处理的情况。

使用`until`会执行循环的其余部分，让GDB在循环后面的第一行代码处暂停。图2-12显示了使用`until`的情况。

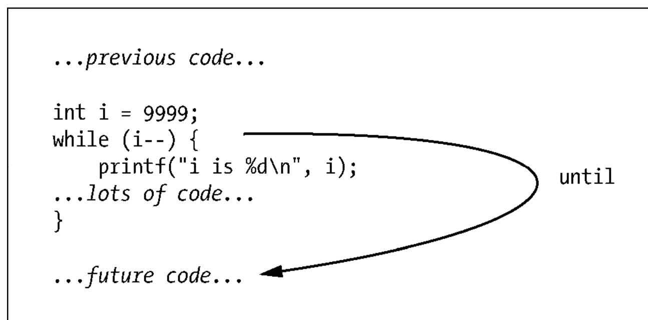


图2-12　`until`将我们带到源代码当前循环体外的下一行

当然，如果GDB在离开循环前遇到一个断点，它就会在那里暂停：如果图2-12中的`printf()`语句处有一个断点，当然要禁用该断点。

GDB使用手册给出了`until`的官方定义：执行程序，直到到达当前循环体外的下一行源代码。

然而，该文档还警告，这可能有点违反直觉。为了说明原因，来看代码清单2-2。

代码清单2-2　*until-anomaly.c*

```
#include <stdio.h>

int main(void)
```

```
{
   int i;

   for (i=0; i<10; ++i)
      printf("hello world!");

   return 0;
}
```

我们将在main()的入口处设置断点，运行程序，并使用until来到达return语句。

```
$ gdb until-anomaly
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x80483b4: file until-anomaly.c, line 7.
(gdb) run
Starting program: until-anomaly

Breakpoint 1, main () at until-anomaly.c:7
7           for (i=0; i<10; ++i)
(gdb) until
8               printf("hello world!");
(gdb) until
7           for (i=0; i<10; ++i)
(gdb) until
10          return 0;
(gdb)
```

可以看出，使用until后，GDB从第7行来到第8行，然后再回到第7行。难道这意味着源代码中的第7行比第8行靠后吗？实际上，底层确实如此。现在也许已经可以猜到答案，因为这似乎是一个常见的主题。GDB最终使用的是机器指令。虽然for结构编写为在主体上方进行循环测试，但是GCC是使用循环主体底部的条件编译程序的。因为该条件与源代码的第7行关联，所以GDB似乎在源代码中反向执行了。事实上，until实际上做的是执行程序，直到它到达内存地址比当前内存地址更高的机器指令，而不是直到到达源代码中一个更大的行号。

在实践中，这种情况可能不是经常出现，但是了解它何时会出现是不错的。另外，通过查看GDB的一些奇怪的行为，可以收集关于编译器如何将源代码转换成机器指令的信息（知道一点这方面的知识没有坏处）。

如果你比较好奇，而且懂得机器的汇编语言，则可以使用GDB的disassemble命令，后面跟着p/x $pc来输出当前位置。[①]这样可以显示until将做的事情。但是这只是一种特殊的情况，在

① 也可以使用GCC的-S选项来查看机器码。使用该选项会产生一个后缀为.s的汇编语言文件，其中显示了编译器产生的代码。注意，这样只会产生汇编语言文件，而不能产生可执行文件。

实际应用中这不是问题。如果在循环的末尾，执行until命令导致回跳到循环顶部，这时只要再次执行until，就可以离开当前的循环。

until命令也可以接受源代码中的位置作为参数。事实上，该命令接受的参数与2.4.1节讨论的break命令相同。回到代码清单2-1中，如果GDB触发了main()入口处的一个断点，那么可以使用下面这些命令方便地使程序一直执行到swap()的入口。

- until 17
- until swap
- until swapflaw.c:17
- until swapflaw.c:swap

2.9.2　在DDD中

本节首先讨论在断点处暂停了DDD后恢复执行的各种方式。

1. 标准操作

DDD的命令工具中对应于next和step的按钮都有。另外，可以分别使用F5和F6功能键执行step和next。

如果要在DDD中使用带参数的next或step，即通过如下代码在GDB中完成要做的事。

```
(gdb) next 3
```

那么需要在DDD的控制台窗口中使用GDB本身。

DDD的命令工具中有一个对应于continue的按钮，但是同样地，如果要执行带参数的continue，需使用GDB控制台。可以单击源窗口来打开continue until here选项，这样可以真正在源代码的那一行处设置临时断点（见2.4.1节）。更精确地说，continue until here意味着“一直继续执行到这一点，但是在任何中间断点处也停止”。

在GDB中，通过带数值参数的next可以获得与finish相同的效果，只是finish或多或少地方便一些。但是在DDD中，使用finish是相当明智的，因为这样只要在命令工具中单击一次鼠标即可。

如果使用的是DDD，则可以使用名为Until on the Command Tool的按钮来执行until，或者单击Program→Until菜单栏，也可以使用键盘快捷键F7。在所有这些选项中，总会发现一种方便的方法！与其他很多GDB命令一样，如果要使用带参数的until，只需要在DDD控制台窗口中直接使用GDB命令。

2. Undo/Redo

正如2.7.6节介绍的，DDD有一个非常有用的Undo/Redo功能。在那一节中，我们介绍了如何撤销不小心的断点删除。这种方法也可以用于Run、Next、Step等动作。

例如，来看图2-13描述的情况。我们在调用swap()时到达了断点，打算执行一个Step操作，但是不小心单击了Next按钮。遇到这种情况，只要单击Undo，就可以将时间倒退回去，如图2-14所示。DDD将当前行的光标显示为轮廓而不是它一般使用的实心绿色，从而提醒你撤销了某些操作。

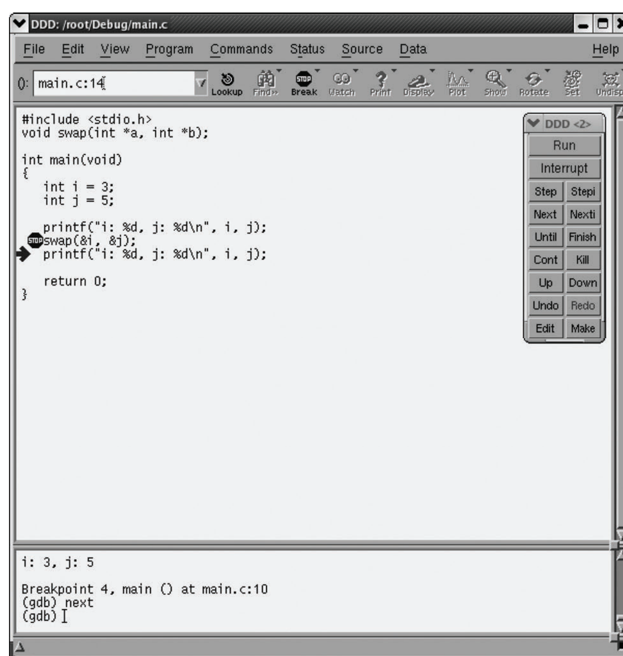


图2-13　不小心按了Next按钮

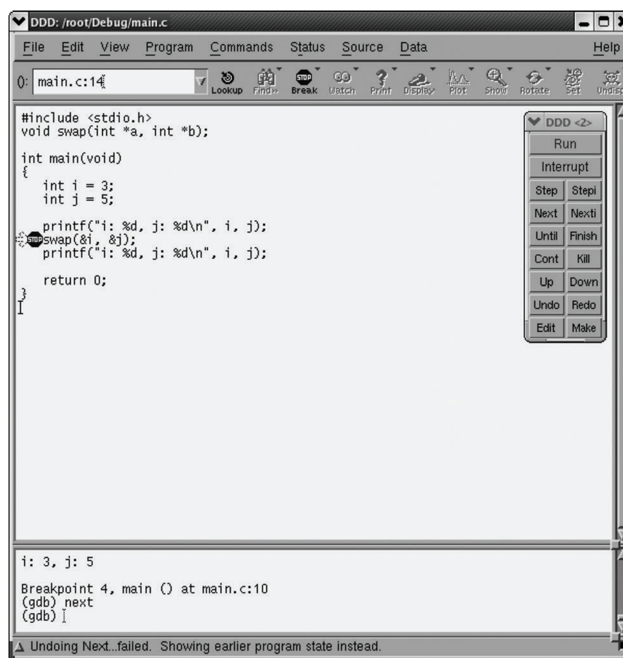


图2-14　单击Undo撤销上一个操作

2.9.3　在Eclipse中

Eclipse中对应于`step`和`next`命令的是Step Into和Step Over图标。Step Into图标在图2-15所示的Debug视图中是可见的（左上方），鼠标指针位置临时导致了图标标签的出现。Step Over图标就在它右边。注意，将要执行的下一个语句是对`get_args()`的调用，因此单击Step Into会导致程序执行的下次暂停发生在该函数的第一个语句中，而选择Step Over意味着下次暂停将在对`process_data()`的调用中。

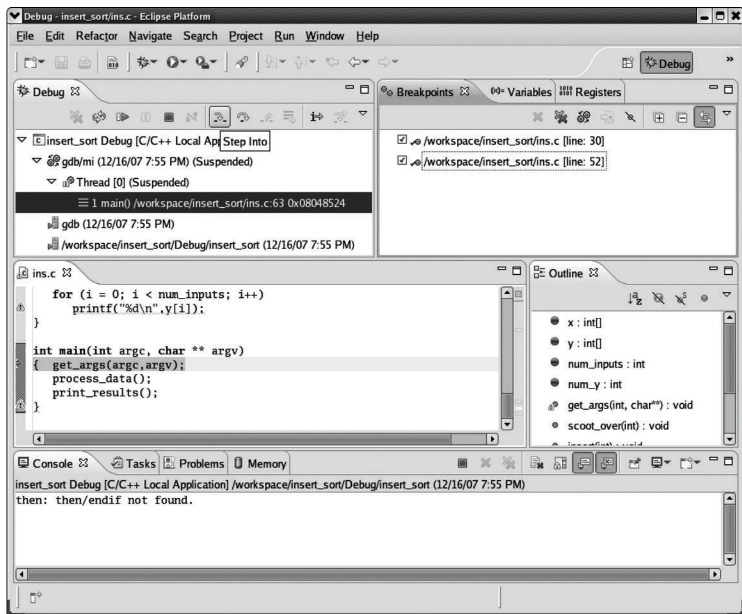


图2-15　Eclipse中的Step Into图标

Eclipse有一个执行`finish`的Step Return图标（在Step Over旁边）。它与`until`的功能完全不一样。在Eclipse中通常使用Run to Line（通过单击目标行然后在源代码窗口中右击来调用）来完成希望使用`until`完成的事。

2.10　条件断点

只要启用了断点，调试器就总是在该断点处停止。然而，有时有必要告诉调试器只有当符合某种条件时才在断点处停止，比如当变量具有我们感兴趣的某个特定的值时。

这类似于监视点的工作方式，但是有一个重要区别。如果怀疑某个变量在某处得到了一个伪值，那么条件断点比监视点更利于作判断。每当该变量的值发生变化时，监视点都会中断。条件断点只会在怀疑有问题的代码处当变量呈现该怀疑值时才中断。从这个意义上讲，当完全不知道变量在何处接受了伪值时，最好使用监视点。这对于全局变量或者在函数之间连续传递的局部变量来说特别有用。但是在其他大多数情况下，位置放得较好的条件断点则更有用、更方便。

2.10.1 GDB

设置条件断点的语法为：

```
break break-args if (condition)
```

其中*break-args*是可以传递给break以指定断点位置的任何参数（如2.4.1节所讨论的），*condition*是2.12.2节定义的布尔表达式。括着*condition*的圆括号是可选的。圆括号会使一些C程序员有宾至如归的感觉，但是你也可能喜欢更整洁的外观。

例如，下面说明了如果用户向程序中键入一些命令行参数，如何在main()处中断[1]：

```
break main if argc > 1
```

条件中断极其有用，尤其适用于索引变量的特定值处出了问题的循环结构。来看如下代码片断：

```
for (i=0; i<=75000; ++i) {
   retval = process(i);
   do_something(retval);
}
```

假设你知道当i为70 000时程序会陷入混乱，要在循环顶部中断，但是不打算用next通过69 999次迭代。这时就非常适合使用条件中断。可以在循环顶部设置一个断点，但是只有当i等于70 000时才中断，代码如下。

```
break if (i == 70000)
```

当然，可以通过键入continue 69999来获得相同的效果，只是这样不大方便。

条件中断也极其灵活，不仅可以测试相等或不相等的变量。在condition中可以使用哪些表达式呢？在有效的C条件语句中几乎可以使用任何表达式。无论使用什么表达式，都需要具有布尔值，即真（非0）或假（0）。包括：

- 相等、逻辑和不相等运算符（<、<=、==、!=、>、>=、&&、||等），例如：

```
break 180 if string==NULL && i < 0
```

- 按位和移位运算符（&、|、^、>>、<<等），例如：

```
break test.c:34 if (x & y) == 1
```

- 算术运算符（+、-、x、/、%），例如：

```
break myfunc if i % (j + 3) != 0
```

[1] 假设声明了argc和argv作为main()的参数。（当然，如果使用不同的名称声明了它们，比如ac和av，那么就使用所声明的名称。）顺便提一下，注意程序总是至少接受一个参数（通过argv[0]指定的程序本身的名称，我们这里没有将它算作“用户参数”）。

- 你自己的函数，只要它们被链接到程序中，例如：

```
break test.c:myfunc if ! check_variable_sanity(i)
```

- 库函数，只要该库被链接到代码中，例如：

```
break 44 if strlen(mystring) == 0
```

由于优先级的次序规则在起作用，因此可能需要使用圆括号将一些结构括起来，比如：

```
(x & y) == 0
```

此外，如果在GDB表达式中使用库函数，而该库不是用调试符号编译的（几乎肯定是这种情况），那么唯一能在断点条件中使用的返回值类型为`int`。换言之，如果没有调试信息，GDB会假设函数的返回值是`int`类型。当这种假设不正确时，函数的返回值会被曲解。

```
(gdb) print cos(0.0)
$1 = 14368
```

然而，类型强制转换也无济于事。

```
(gdb) print (double) cos(0.0)
$2 = 14336
```

如果三角数学知识还没忘的话，就会知道0的余弦是1。

使用非int返回函数

实际上，有一种方式可以在GDB表达式中使用不返回`int`的函数，但是这种方式相当神秘。技巧在于应使用指向函数的恰当数据类型定义一个便于GDB使用的变量。

```
(gdb) set $p = (double (*) (double)) cos
(gdb) ptype $p
type = double (*)()
(gdb) p cos(3.14159265)
$2 = 14368
(gdb) p $p(3.14159265)
$4 = -1
```

如果你的三角数学知识还有一点儿，就会知道3.14159265是π的近似值，π的余弦为−1。

可以对正常断点设置条件将它们转变为条件断点。例如，如果设置了断点3作为无条件断点，但是现在希望添加条件`i==3`，只要键入：

```
(gdb) cond 3 i == 3
```

如果以后要删除条件，但是保持该断点，只要键入：

```
(gdb) cond 3
```

2.10.2 DDD

在DDD中设置条件断点可以通过在控制台窗口中设置GDB语义来实现。或者按如下所示的方式使用DDD。在代码中希望出现条件断点的地方设置正常断点（即非条件断点）。右击并按下红色停止符号来打开一个菜单并选择Properties。这时会出现一个弹出窗口，有一个名为Condition的文本入口框。在该框中键入条件，单击Apply按钮，然后单击Close按钮。断点现在就是条件断点。

这一过程如图2-16所示。我们在断点4上看到了条件`j==0`。顺便说一下，那一行代码上的停止符号现在会包含一个问号，提醒我们这是条件中断。

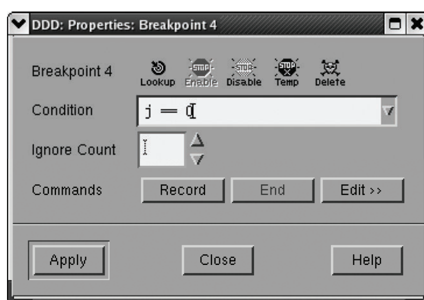


图2-16 在DDD中对断点施加条件

2.10.3 Eclipse

为了使断点成为条件断点，右击该行代码的断点符号，选择Breakpoint Properties...→Common，并在对话框中填充条件。该对话框如图2-17所示。

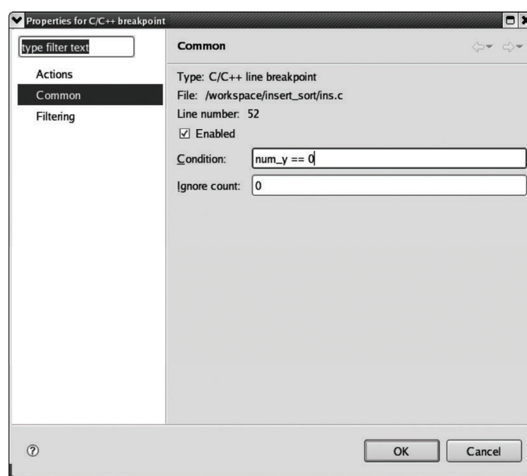


图2-17 在Eclipse中对断点施加条件

2.11　断点命令列表

当GDB遇到断点时，几乎总是要查看某个变量。如果反复遇到同一个断点（与循环内的断点一样），将反复查看相同的变量。让GDB在每次到达某个断点时自动执行一组命令，从而自动完成这一过程，不是很好吗？

事实上，使用“断点命令列表”就可以做这件事。我们将使用GDB的`printf`命令来说明命令列表。虽然本书还没有正式介绍`printf`，但是它在GDB中的工作方式与在C中基本相同，只是圆括号是可选的。

使用`commands`命令设置命令列表。

```
commands breakpoint-number
...
commands
...
end
```

其中*`breakpoint-number`*是要将命令添加到其上的断点的标识符，*`commands`*是每行一个有效GDB命令的列表。逐条输入命令，然后键入`end`表示输入命令完毕。从那以后，每当GDB在这个断点处中断时，它都会执行你输入的任何命令。让我们看一个示例，见代码清单2-3。

代码清单2-3　*fibonacci.c*

```
#include <stdio.h>
int fibonacci(int n);

int main(void)
{
	printf("Fibonacci(3) is %d.\n", fibonacci(3));

	return 0;
}

int fibonacci(int n)
{
	if ( n <= 0 || n == 1 )
		return 1;
	else
		return fibonacci(n-1) + fibonacci(n-2);
}
```

我们打算查看传递给`fibonacci()`的值及其次序。然而，你不想在程序中插入`printf()`语句并重新编译代码。在关于调试的书中这样做很笨拙，是吧？更要命的是，插入代码并重新编译/连接需要花时间，修复了这个特定程序错误后再去掉该代码也需要花时间，尤其是当程序比较大时更是如此。而且，与代码无关的语句会使代码混乱，因此这样使得在调试期间代码难以阅读。

可以使用step单步调试代码，并在每次调用fibonacci()时输出n，但是使用命令列表更好，因为这样就不需要重复键入输出命令。让我们看一下。

首先，在fibonacci()最上方设置一个断点。这个断点将被指定为标识符1，因为它是设置的第一个断点。然后在断点1上设置一个输出变量n的命令。

```
$ gdb fibonacci
(gdb) break fibonacci
Breakpoint 1 at 0x80483e0: file fibonacci.c, line 13.
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>printf "fibonacci was passed %d.\n", n
>end
(gdb)
```

现在运行程序，并查看发生了什么事情。

```
(gdb) run
Starting program: fibonacci

Breakpoint 1, fibonacci (n=3) at fibonacci.c:13
13              if ( n <= 0  ||  n == 1 )
fibonacci was passed 3.
(gdb) continue
Continuing.

Breakpoint 1, fibonacci (n=2) at fibonacci.c:13
13              if ( n <= 0  ||  n == 1 )
fibonacci was passed 2.
(gdb) continue
Continuing.

Breakpoint 1, fibonacci (n=1) at fibonacci.c:13
13              if ( n <= 0  ||  n == 1 )
fibonacci was passed 1.
(gdb) continue
Continuing.

Breakpoint 1, fibonacci (n=0) at fibonacci.c:13
13              if ( n <= 0  ||  n == 1 )
fibonacci was passed 0.
(gdb) continue
Continuing.

Breakpoint 1, fibonacci (n=1) at fibonacci.c:13
```

```
13              if ( n <= 0  ||  n == 1 )
fibonacci was passed 1.
(gdb) continue
Continuing.
Fibonacci(3) is 3.

Program exited normally.
(gdb)
```

这非常接近我们的预期，只是输出太冗长。毕竟，我们已经知道了断点在何处。所幸，可以使用`silent`命令（它需要是命令列表中的第一个命令）使GDB更安静地触发断点。让我们看一下`silent`的实际应用。注意我们如何通过将新命令列表放在我们原来设置的命令列表“上面”来重新定义命令列表。

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>printf "fibonacci was passed %d.\n", n
>end
(gdb)
```

输出如下所示。

```
(gdb) run
Starting program: fibonacci
fibonacci was passed 3.
(gdb) continue
Continuing.
fibonacci was passed 2.
(gdb) continue
Continuing.
fibonacci was passed 1.
(gdb) continue
Continuing.
fibonacci was passed 0.
(gdb) continue
Continuing.
fibonacci was passed 1.
(gdb) continue
Continuing.
Fibonacci(3) is 3.

Program exited normally.
(gdb)
```

现在的输出结果不错。要介绍的最后一个功能是：如果命令列表中的最后一个命令是continue，GDB将在完成命令列表中的命令后继续自动执行程序。

```
(gdb) command 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>printf "fibonacci was passed %d.\n", n
>continue
>end
(gdb) run
Starting program: fibonacci
fibonacci was passed 3.
fibonacci was passed 2.
fibonacci was passed 1.
fibonacci was passed 0.
fibonacci was passed 1.
Fibonacci(3) is 3.

Program exited normally.
(gdb)
```


你可能要在其他程序或者该程序的其他代码行中做这种类型的事情，因此让我们顺势使用GDB的define命令创建宏。

首先，让我们定义宏，命名为print_and_go。

```
(gdb) define print_and_go
Redefine command "print_and_go"? (y or n) y
Type commands for definition of "print_and_go".
End with a line saying just "end".
>printf $arg0, $arg1
>continue
>end
```

要像上述代码那样使用这个宏，应键入：

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>print_and_go "fibonacci() was passed %d\n" n
>end
```

注意，print_and_go的参数之间没有逗号。然后当运行程序时，将得到与以前相同的输出，但是好处是现在可以在代码中的任何地方使用这个宏。而且，可以将宏保在*.gdbinit*文件中，便于

其他程序使用。顺便说一下，尽管这个示例中只有2个参数，但最多允许有10个参数。

键入`show user`可以列出所有宏。

尽管命令列表非常有用，但是将它们与条件中断合并后，将具有更大的威力。使用这种条件输入/输出，甚至可以试着抛掉C语言，只使用GDB作为所选择的编程语言。当然，这只是开个玩笑。

命令与表达式

实际上，2.12.2节列出的任何有效GDB表达式都可以进入命令列表。可以使用库函数，甚至你自己的函数，只要它们被连接到执行文件中即可。可以使用它们的返回值，只要它们返回的值为`int`类型。例如:

```
(gdb) command 2
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>silent
>printf "string has a length of %d\n", strlen(string)
>end
```

DDD中的命令列表类似于DDD中的条件断点。首先，设置断点。右击红色停止符号，并选择Properties。这时会出现一个弹出窗口。右边会有一个大的子窗口（如果看不到这个大子窗口，单击Edit按钮，会使命令窗口出现）。可以将命令都键入到这个窗口中。还有一个Record按钮，右击这个按钮，可以将命令输入GDB控制台中。

Eclipse似乎没有命令列表功能。

2.12　监视点

监视点是一种特殊类型的断点，它类似于正常断点，是要求GDB暂停程序执行的指令。区别在于监视点没有“住在”某一行源代码中，而是指示GDB每当某个表达式改变了值就暂停执行。[①] 该表达式可以非常简单，比如变量的名称:

```
(gdb) watch i
```

它会使得每当i改变值时GDB就暂停。表达式也可以非常复杂。

```
(gdb) watch (i | j > 12) && i > 24 && strlen(name) > 6
```

可以将监视点看做被“附加”在表达式上，当表达式的值改变时，GDB会暂停程序的执行。

虽然管理监视点和断点的方式相同，但是两者之间有一个重要区别。断点与源代码中的一个位置关联。只要代码没有改变，就不存在某行代码“超出作用域”的风险。因为C语言有严格的

① 表达式将在2.12.2节中更全面地介绍。

作用域规则，所以只能监视存在且在作用域内的变量。一旦变量不再存在于调用栈的任何帧中（当包含局部变量的函数返回时），GDB会自动删除监视点。

例如2.5节中名为swapper的程序。在这个程序中，将局部变量c用于临时存储器。在GDB到达定义了c的*swapper.c*的第3行之前，不能在c上设置监视点。此外，如果在c上设置了监视点，一旦GDB从swapper()返回，就会自动删除该监视点，因为c不复存在。

每次变量改变时都让GDB中断，对于被循环紧密包裹的代码或者重复代码来说可能有点讨厌。虽然监视点听上去不错，但是位置选得好的断点可能要有用得多。然而，如果你的一个变量修改了，特别是全局变量，而你又不知道它是在哪里为什么发生了改变，则监视点就是无价的。如果你在处理线程代码，监视点的用处就有限；GDB只能监视单个线程中的变量。第5章对此会进一步介绍。

2.12.1 设置监视点

当变量var存在且在作用域中时，可以通过使用如下命令来设置监视点。

```
watch var
```

该命令会导致每当*var*改变值时GDB都中断。很多人因此而喜欢监视点，觉得它简单方便。然而，还有一些情况要说明。GDB实际上是在*var*的内存位置改变值时中断。一般情况下，是否使用监视点监视变量或变量的地址并没有关系，但是在特殊情况下这一点可能很重要，比如当处理指向指针的指针时。

硬件监视点

当设置监视点时，可以看到一条关于“硬件”监视点的消息:

```
(gdb) watch i
Hardware watchpoint 2: i
```

列出断点时，同样也能看到类似的消息:

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
2   hw watchpoint  keep y              i
```

显然，2是指监视点标识符，i是指被监视的变量，但是“硬件”监视点是指什么呢?

很多平台都有实现监视点的专用硬件。如果有这样的硬件可用，GDB会尝试使用该硬件，因为使用硬件实现任何操作都比较快。

GDB可能无法设置硬件监视点（该平台可能没有必需的硬件，或者硬件正忙于别的事情）。如果是这样，GDB会尝试使用虚拟内存技术实现监视点，这也很快。

如果这两种技术都不可用，GDB会通过软件实现监视点本身，这会非常非常慢。

CPU只能实现有限数量的硬件辅助中断，包括监视点和硬件辅助断点。虽然这个数量与架构有关，但是如果超出了这个限制，GDB就会输出如下消息。

```
Stopped; cannot insert breakpoints.
You may have requested too many hardware breakpoints and watchpoints.
```

如果看到这条消息，就应该删除或禁用部分监视点或硬件辅助断点。

让我们看一个监视点非常有用的示例场景。假设有两个int变量x和y，在代码中的某一处执行p=&y，而你的意图是执行p=&x。这可能会导致y在代码中某处神秘地改变了值。导致程序错误的实际位置可能隐藏得很好，因此断点的用处不会太大。然而，通过设置监视点，可以立刻知道y在何时何处值发生了变化。

监视点的好处甚至还不止这些，它不仅仅限于监视变量。事实上，还可以监视变量的表达式。每当表达式修改值时，GDB都会中断。作为一个示例，来看如下代码。

```
1   #include <stdio.h>
2   int i = 0;
3
4   int main(void)
5   {
6      i = 3;
7      printf("i is %d.\n", i);
8
9      i = 5;
10     printf("i is %d.\n", i);
11
12     return 0;
13  }
```

我们想在每当i大于4时得到通知。因此在main()的入口处放一个断点，以便让i在作用域中，并设置一个监视点以指出i何时大于4。不能在i上设置监视点，因为在程序运行之前，i不存在。因此必须先在main()上设置断点，然后在i上设置监视点。

```
(gdb) break main
Breakpoint 1 at 0x80483b4: file test2.c, line 6.
(gdb) run
Starting program: test2

Breakpoint 1, main () at test2.c:6
```

既然i已经在作用域中了，现在设置监视点并通知GDB继续执行程序。我们完全可以料到，在第9行时会出现i>4的情况。

```
1  (gdb) watch i > 4
2  Hardware watchpoint 2: i > 4
3  (gdb) continue
4  Continuing.
```

```
5  Hardware watchpoint 2: i > 4
6
7  Old value = 0
8  New value = 1
9  main () at test2.c:10
```

果真，GDB在第9行和第10行之间暂停了，这里表达式`i>4`将值从`0`（假）改为了`1`（真）。

可以使用这种方法在控制台窗口中设置监视点，然而，你可能发现使用GUI接口更方便。在源窗口中，找到要在其上设置监视点的变量。这个位置不必是变量第一次出现的地方，可以是它在源代码中出现的任一处。单击该变量以突出显示它，然后单击菜单栏图标中的监视点符号。监视点的设置不会有像断点的红色停止符号那样的标记。为了查看设置的监视点，必须按2.3.2节介绍的方法实际列出所有断点。

在Eclipse中设置监视点的方法如下：在源代码窗口中右击，选择Add a Watch Expression，然后在对话框中填写所需表达式。

2.12.2 表达式

前面我们介绍了通过GDB的`watch`命令使用表达式（expression）的一个示例。该示例表明，有相当多的GDB命令（比如`print`）也接受表达式参数。因此，我们也许应当对它们多进行一点儿介绍。

GDB中的表达式可以包含很多内容：

- 便于GDB使用的变量；
- 程序中的任何在作用域内的变量，比如前面的示例中的`i`；
- 任何种类的字符串、数值或字符常量；
- 预处理器宏（如果程序被编译为包括预处理器调试信息[①]）；
- 条件、函数调用、类型强制转换和所用语言定义的运算符。

因此，如果在调试FORTRAN-77程序，并且想知道变量`i`何时变得大于4，不需要像上一节中那样使用`watch i>4`，而是可以使用`watch i .GT. 4`。

虽然在很多教程和文档中常将C语法用于GDB表达式，但那是由于C和C++的普遍存在的性质；如果使用的是C语言以外的语言，GDB表达式就得用该语言的元素中构建。

① 在编写本书时，GNU GDB官方使用手册指出预处理器宏不能用在表达式中，然而情况并非如此。如果使用GCC的`-g3`选项编译，就能在表达式中使用预处理器宏。

第 3 章

检查和设置变量

第1章介绍过，在GDB中可以使用print命令输出变量的值，在DDD和Eclipse中可以通过将鼠标指针移到源代码中任何地方的变量的实例上来输出变量的值。但是GDB和GUI都提供了更强大的检查变量和数据结构的方法，这是本章将重点介绍的内容。

3.1 主要示例代码

下面是二叉树的一种直观实现（还没有顾及到效率、模块化等）。

```
// bintree.c: routines to do insert and sorted print of a binary tree

#include <stdio.h>
#include <stdlib.h>
struct node {
   int val;             // stored value
   struct node *left;   // ptr to smaller child
   struct node *right;  // ptr to larger child
};

typedef struct node *nsp;

nsp root;

nsp makenode(int x)
{
   nsp tmp;

   tmp = (nsp) malloc(sizeof(struct node));
   tmp->val = x;
   tmp->left = tmp->right = 0;
   return tmp;
}

void insert(nsp *btp, int x)
```

```
{
   nsp tmp = *btp;

   if (*btp == 0) {
      *btp = makenode(x);
      return;
   }

   while (1)
   {
      if (x < tmp->val) {

         if (tmp->left != 0) {
            tmp = tmp->left;
         } else {
            tmp->left = makenode(x);
            break;
         }

      } else {

         if (tmp->right != 0) {
            tmp = tmp->right;
         } else {
            tmp->right = makenode(x);
            break;
         }

      }
   }
}

void printtree(nsp bt)
{
   if (bt == 0) return;
   printtree(bt->left);
   printf("%d\n",bt->val);
   printtree(bt->right);
}


int main(int argc, char *argv[])
{  int i;

   root = 0;
   for (i = 1; i < argc; i++)
```

```
        insert(&root, atoi(argv[i]));
     printtree(root);
  }
```

在各个节点上，左子树的所有元素都小于给定节点中的值，右子树的所有元素都大于等于给定节点中的值。函数`insert()`创建了一个新节点，并将它放在树中恰当的位置。函数`printtree()`按数字升序显示任意子树的元素，而`main()`运行一个测试，输出整个排序后的数组。[①]

对于这里的调试示例，假设不小心在`insert()`中把对`makenode()`进行第二次调用的代码错写成：

```
tmp->left = makenode(x);
```

而不是

```
tmp->right = makenode(x);
```

如果运行这种带程序错误的代码，马上就会出错。

```
$ bintree 12 8 5 19 16
16
12
```

让我们探索一下调试工具中的各种检查命令如何帮助尽快找到程序错误。

3.2　变量的高级检查和设置

我们这里的树示例复杂性更高，因此需要更高级的方法。下面将介绍部分方法。

3.2.1　在GDB中检查

在以前的章节中使用过GDB的基本`print`命令。在这里如何使用它呢？主要工作显然是在`insert()`中完成，因此该方法将是一个好的起始点。当仍然在该函数的`while`循环中运行GDB时，可以在每次遇到断点时执行一组（3个）GDB命令。

```
(gdb) p tmp->val
$1 = 12
(gdb) p tmp->left
$2 = (struct node *) 0x8049698
(gdb) p tmp->right
$3 = (struct node *) 0x0
```

（在第1章介绍过，GDB的输出被标记为`$1`、`$2`等，这些数量统称为*值历史*。我们将在3.4.1节进一步讨论它们。）

在这里将发现，`tmp`当前指向的节点包含12，有一个非0左指针，而不是为0的右指针。当然，

① 顺便说一下，注意第12行中的类型定义`nsp`。这代表*node struct pointer*，但是我们的出版商认为它是*No Starch Press*。

左指针的实际值，即实际内存地址，可能与这里介绍的内容没有直接关系，但是指针是否为0这一事实是重要的。关键是这里目前看到的是低于12的左子树，而不是右子树。

● 第一个改进：输出完整的结构

每次到达一个断点时都要键入这3个print命令会非常费力。我们可以只用一个print命令来完成，用法如下。

```
(gdb) p *tmp
$4 = {val = 12, left = 0x8049698, right = 0x0}
```

由于tmp指向该结构，因此*tmp是这个结构本身，GDB向我们显示了完整的内容。

● 第二个改进：使用GDB的display命令

键入上面的p *tmp能节省时间和精力。每次遇到断点时，只需要键入一个GDB命令，而不是3个。但是，如果知道会在每次遇到断点时都键入这个命令，那么使用GDB的display命令（简写为disp）能节省更多时间和精力。这个命令要求GDB在执行中每次有暂停（由于有断点，使用next或step命令等）时就输出指定条目。

```
(gdb) disp *tmp
1: *tmp = {val = 12, left = 0x8049698, right = 0x0}
(gdb) c
Continuing.

Breakpoint 1, insert (btp=0x804967c, x=5) at bintree.c:37
37            if (x < tmp->val)  {
1: *tmp = {val = 8, left = 0x0, right = 0x0}
```

正如在这里所见的，GDB在遇到断点以后自动输出*tmp，因为执行了display命令。

当然，只有当变量在作用域中时，才会将显示列表中的变量显示出来。

● 第三个改进：使用GDB的commands命令

假设希望在给定节点上时查看子节点的值。第1章介绍过GDB的commands命令，可以按如下代码使用。

```
(gdb) b 37
Breakpoint 1 at 0x8048403: file bintree.c, line 37.
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>p tmp->val
>if (tmp->left != 0)
 >p tmp->left->val
 >else
 >printf "%s\n", "none"
 >end
```

```
>if (tmp->right != 0)
 >p tmp->right->val
 >else
 >printf "%s\n", "none"
 >end
>end
```

注意，在这个示例中，GDB的`print`命令有一个更强大的相关命令——`printf()`，它的格式与C语言同名命令相似。

下面是得到的GDB会话的采样。

```
Breakpoint 1, insert (btp=0x804967c, x=8) at bintree.c:37
37            if (x < tmp->val)
$7 = 12
none
none
(gdb) c
Continuing.

Breakpoint 1, insert (btp=0x804967c, x=5) at bintree.c:37
37            if (x < tmp->val)
$6 = 12
$7 = 8
none
(gdb) c
Continuing.

Breakpoint 1, insert (btp=0x804967c, x=5) at bintree.c:37
37            if (x < tmp->val)
$8 = 8
none
none
```

当然，使用标签等可以使输出更易读。

- 第四个改进：使用GDB的`call`命令

调试中的一个常见方法是隔离出现问题的第一个数据项。在这个例子中，可以通过在每次完成对`insert()`的调用时输出整个树来完成这件事。由于无论如何源文件中都有一个函数——`printtree()`可以做这件事，因此可以简单地在源代码中调用`insert()`之后马上增加对这个函数的调用。

```
for (i = 1; i < argc; i++)  {
   insert(&root,atoi(argv[i]));
   printtree(root);
}
```

然而，从各种角度来看这样可能不理想。比如，这可能意味着必须花时间来编辑源代码文件并重新编译。前者会分散注意力，后者在程序比较大的情况下比较费时间。毕竟，这是试图通过使用调试器避免的事情。

相反，从GDB中做这件事就比较好。可以通过GDB的`call`命令做这件事。例如，可以在第57行（即`insert()`的末尾）设置一个断点，然后执行如下代码。

```
(gdb) commands 2
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>printf "*********** current tree ***********"
>call printtree(root)
>end
```

得到的GDB会话片断如下：

```
Breakpoint 2, insert (btp=0x8049688, x=12) at bintree.c:57
57 }
******** current tree ********
12
(gdb) c
Continuing.

Breakpoint 2, insert (btp=0x8049688, x=8) at bintree.c:57
57 }
*********** current tree ***********
8
12
(gdb) c
Continuing.

Breakpoint 2, insert (btp=0x8049688, x=5) at bintree.c:57
57 }
*********** current tree ***********
5
8
12
(gdb) c
Continuing.

Breakpoint 2, insert (btp=0x8049688, x=19) at bintree.c:57
57 }
*********** current tree ***********
19
12
```

3

注意，这一结果表示引起问题的第一个数据项是19。有了这一信息可以很快直接瞄准程序错误。使用同样的数据重新运行程序，在`insert()`的开头设置一个断点，但是这次使用条件`x == 19`，然后研究那里发生了什么事情。

可以动态地修改给定断点的命令集，或者简单地通过重新定义一个空集合来取消该命令集。

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>end
```

3.2.2　在DDD中检查

现在你已经知道，可以从DDD的DDD控制台窗口中调用任何GDB命令。但是DDD 的一个真正的优点是在DDD中运行很多GDB命令更方便，在有些情况下，DDD能够执行GDB无法提供的强有力的操作，比如显示下面描述的链接数据结构。

正如前面几章所提到的，在DDD中检查变量的值是非常方便的：只要将鼠标指针移到源代码窗口中变量的任何实例上。但是还有其他很多方法也值得使用，尤其是对于使用链接数据结构的程序。我们将通过使用本章前面的二叉树示例来说明。

前面介绍过GDB的`display`命令：

```
(gdb) disp *tmp
```

这里，`tmp`指向结构的内容在每次遇到断点时自动输出，否则会导致暂停执行程序。由于`tmp`是指向这棵树中当前节点的指针，因此这种自动输出有助于监视遍历这棵树时的进度。

DDD中对应的功能是Display命令。如果在源代码窗口中右击变量的任一实例，比如本例中的`root`，则会弹出一个图3-1所示的菜单。可以看到，该菜单中有一些关于查看`root`的选项。选项`Print root`和`Print *root`的工作方式与它们的GDB对应命令完全相同，事实上它们的输出出现在DDD的控制台中（在这里回显/输入GDB命令）。但是对于目前的情况，最有趣的选项是`Display*root`。遇到源代码的第48行的断点后，选择该选项的结果如图3-2所示。

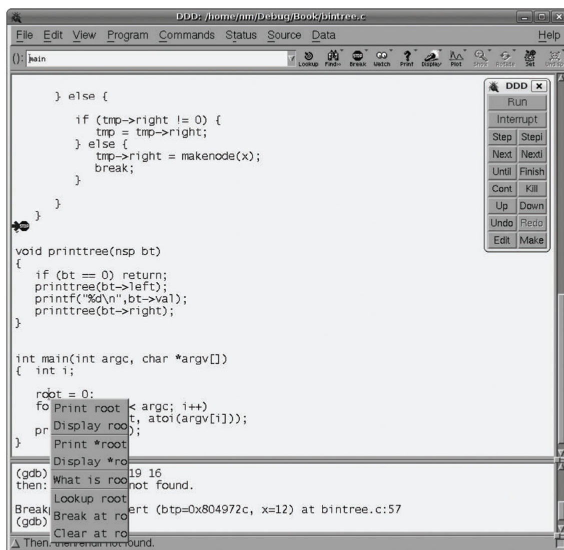


图3-1　查看变量的弹出窗口

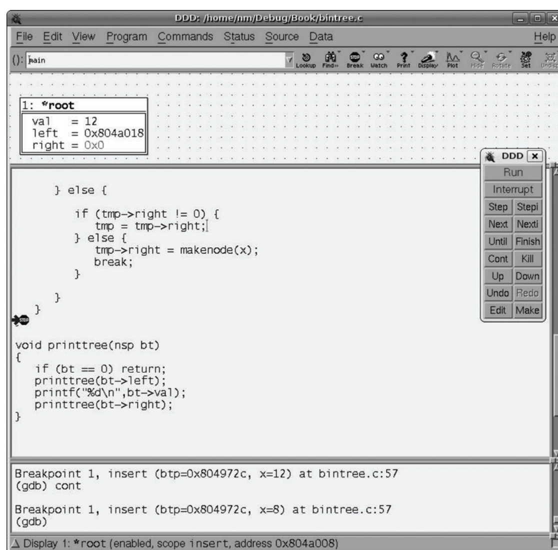


图3-2　节点的显示

这时出现一个新的DDD窗口——数据窗口，其中的节点对应于`root`。到目前为止，这个命令无非只是像图形化的GDB的`display`命令。但是这里真正的好处是可以跟踪树链接。比如，要跟踪树的左分支，只要右击显示的根节点的`left`字段。（这时不要对`right`节点这样做，因为链接为0。）然后在弹出菜单中选择`Display *()`选项，现在DDD如图3-3所示。因此，DDD呈现了这棵树（或者树的一部分）的图，就像在黑板上画出来的一样，非常酷！

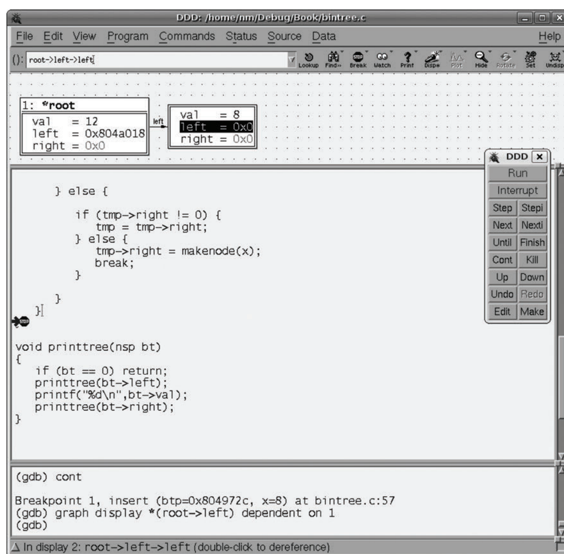


图3-3　跟踪链接

每当节点内容改变时，现有节点内容的显示会自动更新。每次一个链接从0改变为非0时，都可以右击它以显示新节点。

显然，数据窗口很快会变得混乱。你可以通过单击并拖动该窗口右下方的小方框来展开窗口，但是更好的方法是在启动DDD前就预见到这种情况。如果用separate命令行选项调用DDD：

```
$ ddd --separate bintree
```

那么会出现一些单独的窗口——源代码窗口、控制台窗口和数据窗口，这些窗口的大小都可以重新调整。

如果要在DDD会话期间删除数据窗口的部分或全部内容，有很多方式可以做到这一点。例如，可以右击一个条目，然后选择Undisplay选项。

3.2.3　在Eclipse中检查

与DDD一样，要在Eclipse中检查标量变量，只要将鼠标指针移到源代码窗口中的变量的任何实例上即可。注意，这必须是一个独立的标量，而不是在struct中这样，如图3-4中所示。这里我们成功地查询了x的值，但要是将鼠标指针指向同一行中tmp->val的val部分，就不会显示那里是什么。

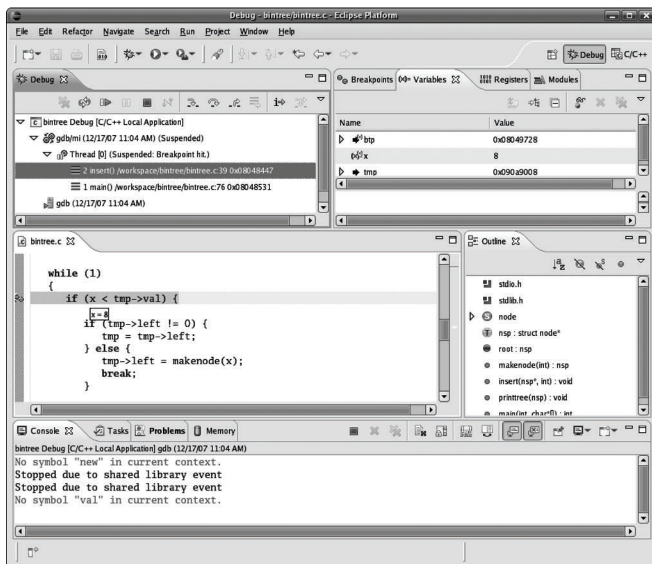


图3-4　在Eclipse中检查标量变量

这时，可以使用Eclipse的Variables视图，在图3-5所示界面的右上方可以看到。单击tmp旁边的三角形来将它向下指，然后向下滚动一行，这时将发现显示了tmp->val。（显示为包含12）。

继续进行这一过程。当单击了left旁边的三角形以后，将看到图3-6中显示的屏幕，在其中可以看到tmp->left->val为8。

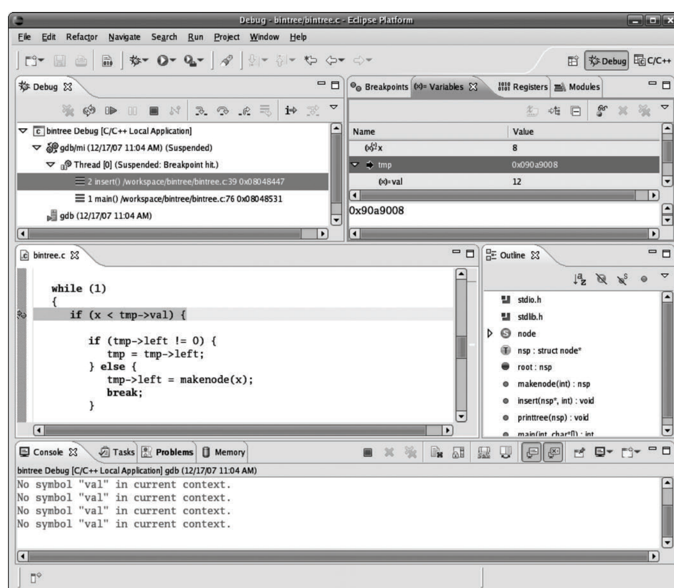


图3-5　在Eclipse中检查`struct`字段

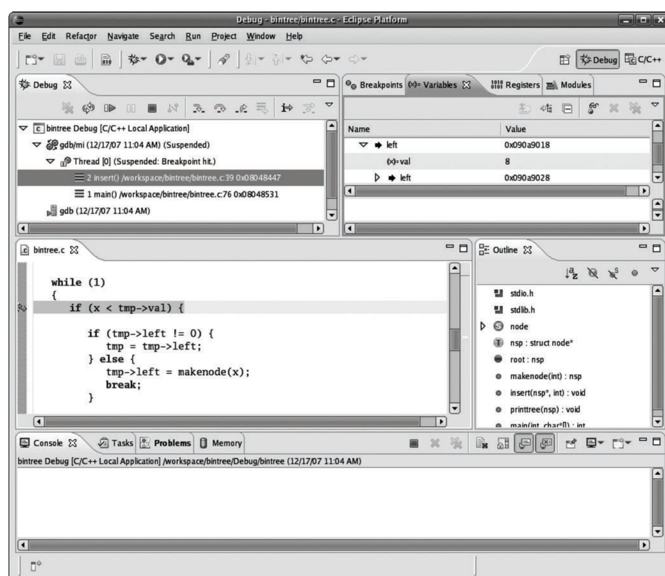


图3-6　在Eclipse中跟踪指针链接

默认情况下，Variables视图不显示全局变量。在这里的程序中，有一个变量`root`。可以将该变量添加到Variables视图中：在该视图中右击，选择Add Global Variables，在出现的弹出式窗口中检查`root`的框，然后单击OK按钮。

3.2.4 检查动态数组

正如第1章所讨论的，在GDB中，可以输出整个数组，比如对于如下数组：

```
int x[25];
```

方法是通过键入：

```
(gdb) p x
```

但是，如果是动态创建的数组会是什么样呢？比如：

```
int *x;
...
x = (int *) malloc(25*sizeof(int));
```

如果要在GDB中输出数组，就不能键入：

```
(gdb) p x
```

这个命令只可以简单打印数组地址。也不能键入：

```
(gdb) p *x
```

这样只会输出数组的一个元素——`x[0]`。仍然可以像在命令`p x[5]`中那样输出单个元素，但是不能简单地在`x`上使用`print`命令输出整个数组。

1. GDB中的解决方案

在GDB中，可以通过创建一个人工数组（artificial array）来解决这个问题。来看如下代码。

```
1  int *x;
2
3  main()
4  {
5     x = (int *) malloc(25*sizeof(int));
6     x[3] = 12;
7  }
```

然后可以如下执行：

```
Breakpoint 1, main () at artif.c:6
6          x = (int *) malloc(25*sizeof(int));
(gdb) n
7          x[3] = 12;
(gdb) n
8       }
(gdb) p *x@25
$1 = {0, 0, 0, 12, 0 <repeats 21 times>}
```

可以看到，一般形式为：

```
*pointer@number_of_elements
```

GDB还允许在适当的时候使用类型强制转换，比如：

```
(gdb) p (int [25]) *x
$2 = {0, 0, 0, 12, 0 <repeats 21 times>}
```

2. DDD中的解决方案

与惯常一样，可以利用GDB方法，本节中是通过DDD控制台使用人工数组。

另一个选项是输出或显示内存的范围（参见3.2.7节）。

3. Eclipse中的解决方案

这里可以使用Eclipse的Display As Array命令。

例如，让我们稍微扩展一下前面的示例。

```
1  int *x;
2
3  main()
4  {
5     int y;
6     x = (int *) malloc(25*sizeof(int));
7     scanf("%d%d",&x[3],&x[8]);
8     y = x[3] + x[8];
9     printf("%d\n",y);
10 }
```

假设目前在给y赋值。首先通过右击Variable视图并选择x来将x放到该视图中。然后将再次在Variables视图中右击x，并选择Display As Array。在得到的弹出框中，填充Start Index和Length字段，比如分别填写为0和25来显示整个数组。屏幕现在如图3-7所示。可以看到Variable视图中的数

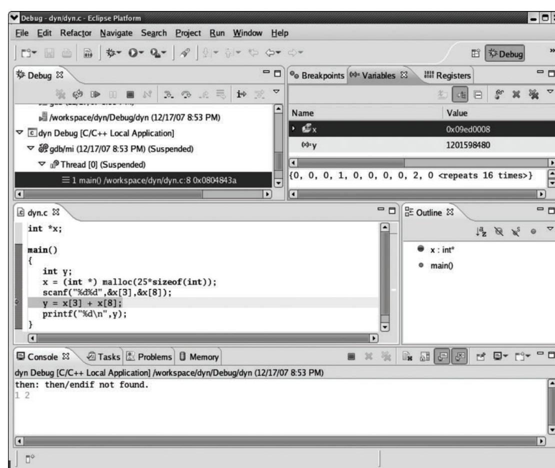


图3-7　在Eclipse中显示一个动态数组

组，显示为：

```
(0,0,0,1,0,0,0,0,2,0,<repeats 16 times>)
```

值1和2来自我们对程序的输入，参见控制台视图。

3.2.5　C++代码的情况

为了说明C++代码的情况，下面是前面用过的二叉树示例的C++版本。

```
// bintree.cc: routines to do insert and sorted print of a binary tree in C++

#include <iostream.h>

class node {
   public:
      static class node *root;  // root of the entire tree
      int val;  // stored value
      class node *left;  // ptr to smaller child
      class node *right;  // ptr to larger child
      node(int x);  // constructor, setting val = x
      static void insert(int x);  // insert x into the tree
      static void printtree(class node *nptr);  // print subtree rooted at *nptr
};

class node *node::root = 0;


node::node(int x)
{
   val = x;
   left = right = 0;
}


void node::insert(int x)
{
   if (node::root == 0)  {
      node::root = new node(x);
      return;
   }
   class node *tmp=root;
   while (1)
   {
      if (x < tmp->val)
      {

         if (tmp->left != 0) {
```

```
            tmp = tmp->left;
         } else {
            tmp->left = new node(x);
            break;
         }

      } else {

         if (tmp->right != 0) {
            tmp = tmp->right;
         } else {
            tmp->right = new node(x);
            break;
         }

      }
   }
}


void node::printtree(class node *np)
{
   if (np == 0) return;
   node::printtree(np->left);
   cout << np->val << endl;
   node::printtree(np->right);
}
int main(int argc, char *argv[])
{
   for (int i = 1; i < argc; i++)
      node::insert(atoi(argv[i]));
   node::printtree(node::root);
}
```

仍然与以前一样编译，并且一定要让编译器在可执行文件中保留符号表。[①]

相同的GDB命令也能起作用，但是输出略有不同。例如，再次输出`insert()`中的`tmp`指向对象的内容，可以得到如下输出。

```
(gdb) p *tmp
$6 = {static root = 0x8049d08, val = 19, left = 0x0, right = 0x0}
```

① 不过，在这方面可能产生各种各样的问题。见GDB手册中关于可执行文件格式的内容。这里的示例使用G++编译器（GCC的C++包装器），用`-g`选项指定应当保留符号表。我们还尝试过`-gstabs`选项，虽然它也能起作用，但是产生的结果不太理想。

这类似于C程序的情况，只是现在还输出了`static`变量`node::root`的值（它应当是该变量值，因为它是该类的一部分）。

当然，必须记住，GDB需要根据与C++使用的相同作用域规则来指定变量。例如：

```
(gdb) p *root
Cannot access memory at address 0x0
(gdb) p *node::root
$8 = {static root = 0x8049d08, val = 12, left = 0x8049d18, right = 0x8049d28}
```

我们需要通过它的全名`node::root`指定`root`。

虽然GDB和DDD没有内置类浏览器，但是GDB的`ptype`命令可以很方便地快速浏览类或结构（struct）的结构（structure），例如：

```
(gdb) ptype node
type = class node {
  public:
    static node *root;
    int val;
    node *left;
    node *right;

    node(int);
    static void insert(int);
    static void printtree(node*);
}
```

在DDD中，可以右击类或变量名，然后在弹出菜单中选择What Is来得到相同的信息。

Eclipse中有Outline视图，因此可以轻松地以这种方式得到类信息。

3.2.6　监视局部变量

在GDB中，可以通过调用`info locals`命令列出当前栈帧中所有局部变量的值。

在DDD中，甚至可以通过单击Data→Display Local Variables来显示局部变量。这会导致DDD数据窗口一个区域专门用来显示局部变量（当单步调试程序时会更新）。GDB中似乎没有做这件事的直接方式，不过可以逐个断点地做这件事：为希望自动输出局部变量的每个断点在`commands`例程中加上`info locals`命令。

可以看到，Eclipse在Variables视图中显示局部变量。

3.2.7　直接检查内存

在有些情况下，可能希望检查给定地址的内存，而不是通过变量的名称。GDB为这种目的提供了`x`命令（表示examine）。在DDD中，选择Data→Memory，指定起点和字节数，并在Print和Display之间做选择。Eclipse有一个Memory视图，在其中可以创建Memory Monitors。

这主要适用于汇编语言环境，第8章将详细讨论。

3.2.8 print和display的高级选项

print和display命令允许指定可选的格式。例如：

```
(gdb) p/x y
```

会以十六进制格式显示变量，而不是十进制格式。其他常用的格式为c表示字符（character），s表示字符串（string），f表示浮点（floating-point）。

可以临时禁用某个显示项。例如，

```
(gdb) dis disp 1
```

临时禁用显示列表中的条目1。如果不知道条目号，可以通过info disp命令检查。要重新启用条目，使用enable，例如，

```
(gdb) enable disp 1
```

要完全删除显示的条目，使用undisplay，例如，

```
(gdb) undisp 1
```

3.3 从 GDB/DDD/Eclipse 中设置变量

在有些情况下，在单步调试程序的过程中间想要使用调试器设置变量的值。这样可以快速地判断某个程序错误的各种来源都会产生怎样的结果。

在GDB中，值的设置非常容易，例如，

```
(gdb) set x = 12
```

会将x的当前值改成12。

DDD中似乎没有用鼠标来设置这样的值的方式，但是同样，可以通过DDD控制台窗口在DDD中执行任何GDB命令。

在Eclipse中，打开Variables视图，右击要设置其值的变量，并选择Change Value。然后可以在弹出窗口中填充新值。

可以通过GDB的set args命令设置程序的命令行参数。然而，与第1章介绍的方法相比，这种方法并没有优势，第1章的方法只要在调用GDB的run命令时使用新参数即可。这两种方法是完全等价的。例如，如下代码：

```
(gdb) set args 1 52 19 11
```

并不会立即将argv[1]改成1，argv[2]改成52，等等，直到下次执行run命令时才会发生这些变化。

GDB有用来检查当前函数参数的info args命令。当单击Data→Display Arguments时，DDD会提供这一功能。

3.4　GDB 自己的变量

除了在程序中声明的变量外，GDB还为其他变量提供了一些机制。

3.4.1　使用值历史

GDB的print命令的输出值被标为$1、$2等，这些值统称为值历史。在将来执行print命令时使用这样的值历史会比较方便。

例如，考虑3.1节的bintree示例。GDB的部分会话可能如下所示。

```
(gdb) p tmp->left
$1 = (struct node *) 0x80496a8
(gdb) p *(tmp->left)
$2 = {val = 5, left = 0x0, right = 0x0}
(gdb) p *$1
$3 = {val = 5, left = 0x0, right = 0x0}
```

这里发生的事情是，当我们输出指针tmp->left的值后发现它为非0，我们决定输出这个指针指向的内容。我们输出这样的内容两次。第一次采用方便的方式，第二次通过值历史。

在第三次输出中，我们引用值历史中的$1。如果没有进行常规输出，则可以使用特殊的值历史变量$。

```
(gdb) p tmp->left
$1 = (struct node *) 0x80496a8
(gdb) p *$
$2 = {val = 5, left = 0x0, right = 0x0}
```

3.4.2　方便变量

假设有一个指针变量p，它在不同的时候指向链表中的不同节点。在调试会话期间，可能希望记录特定节点的地址，比如，因为你希望重新检查调试过程中节点在不同时候的值。当p首次到达该节点时，可以这样做：

```
(gdb) set $q = p
```

此后执行的命令如下：

```
(gdb) p *$q
```

这里的变量$q称为方便变量（convenience variable）。

方便变量会根据C规则改变值。例如，来看代码：

```
int w[4] = {12,5,8,29};

main()
```

```
{
   w[2] = 88;
}
```

在GDB中可能做一些类似如下的事情。

```
Breakpoint 1, main () at cv.c:7
7          w[2] = 88;
(gdb) n
8       }
(gdb) set $i = 0
(gdb) p w[$i++]
$1 = 12
(gdb)
$2 = 5
(gdb)
$3 = 88
(gdb)
$4 = 29
```

为了理解这里发生了什么，回顾一下，如果我们简单地在GDB中按下ENTER键而没有发出命令，GDB就会将这一操作看作重复上一个命令的请求。在上面的GDB会话中，保持按下ENTER键，就意味着在要求GDB重复如下命令。

```
(gdb) p w[$i++]
```

这不仅意味着会输出一个值，而且方便变量`$i`也会递增。

说明 几乎可以为方便变量选择任何名称，不过有一些例外。例如，不能给方便变量起名为`$3`，因为那是为值历史保留的名称。另外，如果用的是汇编语言，则不应使用寄存器名称。例如，对于Intel x86系列机器，有一个寄存器名为EAX，在GDB中它被称为`$eax`，如果在汇编语言层上工作，则不要选择这个名称作为方便变量。

第 4 章

程序崩溃处理

有人说C语言是低级语言。这一部分是因为应用程序的内存管理大部分需要由程序员来实现。虽然这种方法非常有用，但是也给程序员增加了很多麻烦。

也有人说，C语言是相对较小且容易学习的语言。然而，只有不考虑标准C语言库的典型实现时，C语言才比较小，实际上这个库相当庞大。而且很多程序员认为C语言是易用语言，那是因为他们还没有遇到指针。

一般而言，程序错误会导致下面两件事的发生。

- 导致程序做一些程序员没有打算做的事。这样的程序错误通常因为逻辑缺陷，比如在第3章的数字排序程序中，将节点放在了树的错误分支上。到目前为止我们一直在集中介绍这种程序错误。
- 导致程序“爆炸”或“崩溃”。这些程序错误通常与指针的误操作或误用有关。这是本章将介绍的程序错误类型。

4.1 背景资料：内存管理

当程序崩溃时到底发生了什么事？我们这里将解释一下，并找出它与产生崩溃的程序错误有什么关系。

4.1.1 为什么程序会崩溃

用编程界的行话说，当某个错误导致程序突然和异常地停止执行时，程序崩溃。迄今最常见的导致程序崩溃的原因是试图在未经允许的情况下访问一个内存位置。硬件会感知这件事，并执行对操作系统（OS）的跳转。在本书主要使用的Unix系列的平台上，操作系统一般会宣布程序导致了段错误（seg fault），并停止程序的执行。在微软的Windows系统上，对应的术语是一般保护错误（general protection fault）。无论是哪个名称，硬件都必须支持虚拟内存，而且操作系统必须使用虚拟内存才会发生这个错误。虽然这是如今的通用计算机的标准，但是读者应记住，专用的小型计算机一般没有这种情况，比如用来控制机器的嵌入式计算机。

为了有效地使用GDB/DDD处理段错误，重要的是要真正理解内存访问错误是如何发生的。在下面几页中，我们简要介绍一下虚拟内存（VM）在程序执行期间所起的作用。我们将把焦点

放在虚拟内存问题与段错误之间的关系上。因此，即使你在计算机课程中学习过虚拟内存，本节介绍的内容仍然可以帮助你处理调试工作中的段错误。

4.1.2 内存中的程序布局

正如前面提到的，当程序中有内存访问问题时，会发生段错误。为了讨论这件事，重要的是先理解程序在内存中是如何布局的。

在Unix平台上，为程序分配的虚拟地址的布局通常类似于图4-1。

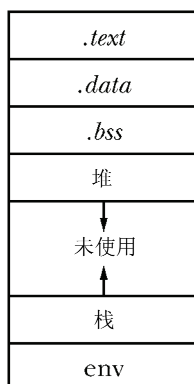


图4-1 程序内存布局

这里虚拟地址0在最下方，箭头显示了其中两个组件（堆和栈）的增长方向，当它们增长时，消耗掉未使用的自由区域。各个部分的作用如下所示。

- 文本区域，由程序源代码中的编译器产生的机器指令组成。例如，每行C代码通常会转换成两到三条机器指令，所有结果指令的集合组成了可执行文件的文本部分。这个部分的正式名称是*.text*。

  这一组件包括静态链接代码，包括做初始化工作然后调用`main()`的系统代码*/usr/lib/crt0.o*。

- 数据区域，包含在编译时分配的所有程序变量，即全局变量。

  实际上，这个区域由各种各样的子区域组成。第一个子区域称为*.data*，由初始化过的变量组成，即在如下所示的声明中给出的变量。

```
int x = 5;
```

  还有一种用于存放未初始化数据的*.bss*区域，在如下所示的声明中给出。

```
int y;
```

- 当程序在运行时从操作系统中请求额外的内存时（例如，当在C语言中调用`malloc()`时，或者在C++中调用`new`结构时），请求的内存在名为堆的区域中分配。如果堆空间不够，可以通过调用`brk()`来扩展堆（这正是`malloc()`及相关函数所做的事情）。

- 栈区域，是用来动态分配数据的空间。函数调用的数据（包括参数、局部变量和返回地址）都存储在栈上。每次进行函数调用时栈都会增长，每次函数返回到其调用者时栈都会收缩。
- 图4-1中没有显示程序的动态链接代码，它的位置与平台有关，但是它确实在某个地方存在。

让我们稍微探究一下。来看如下代码。

```
int q[200];

int main( void )
{
   int i, n, *p;
   p = malloc(sizeof(int));
   scanf("%d", &n);
   for (i = 0; i < 200; i++)
      q[i] = i;

   printf("%x  %x  %x  %x  %x\n", main, q, p, &i, scanf);

        return 0;
}
```

虽然该程序本身作用并不大，但是我们将它编写成一个工具来非正式地探索虚拟地址空间的布局。出于这个目的，让我们运行以下程序。

```
% a.out
5
80483f4  80496a0  9835008  bfb3abec  8048304
```

从运行结果可以看到文本区域、数据区域、堆、栈和动态链接函数的大体位置分别是`0x080483f4`、`0x080496a0`、`0x09835008`、`0xbfb3abec`和`0x08048304`。

可以通过查看这一过程的*maps*文件来得到程序在Linux上的精确内存布局情况。这个过程号是21111，因此我们将查看相应的文件*/proc/21111/maps*。

```
$ cat /proc/21111/maps
009f1000-009f2000 r-xp 009f1000 00:00 0          [vdso]
009f2000-00a0b000 r-xp 00000000 08:01 4116750    /lib/ld-2.4.so
00a0b000-00a0c000 r-xp 00018000 08:01 4116750    /lib/ld-2.4.so
00a0c000-00a0d000 rwxp 00019000 08:01 4116750    /lib/ld-2.4.so
00a0f000-00b3c000 r-xp 00000000 08:01 4116819    /lib/libc-2.4.so
00b3c000-00b3e000 r-xp 0012d000 08:01 4116819    /lib/libc-2.4.so
00b3e000-00b3f000 rwxp 0012f000 08:01 4116819    /lib/libc-2.4.so
00b3f000-00b42000 rwxp 00b3f000 00:00 0
08048000-08049000 r-xp 00000000 00:16 18815309   /home/matloff/a.out
```

```
08049000-0804a000 rw-p 00000000 00:16 18815309   /home/matloff/a.out
09835000-09856000 rw-p 09835000 00:00 0          [heap]
b7ef8000-b7ef9000 rw-p b7ef8000 00:00 0
b7f14000-b7f16000 rw-p b7f14000 00:00 0
bfb27000-bfb3c000 rw-p bfb27000 00:00 0          [stack]
```

你不需要理解所有这些代码。重点是在该显示结果中，可以看到文本和数据区域（从文件*a.out*中），以及堆和栈。从中也可以看到C库（对于调用`scanf()`、`malloc()`和`printf()`）被放在何处（从文件*/lib/libc-2.4.so*）。此外还应识别出一个权限字段，其格式类似于使用*ls*显示的文件权限，比如表示`rw-p`这样的权限。后者很快就会介绍。

4.1.3　页的概念

图4-1所示的虚拟地址空间概念上从0延伸到2^w-1，其中w是机器上按位表示的单词大小。当然，程序通常只会使用该空间的很少一部分，操作系统可能保留空间的一部分给自己用。但是程序员编写的代码可以通过指针在该范围内的任何地方生成地址。通常这样的地址会是错误的，原因在于程序中有程序错误！

虚拟地址空间是通过组织成称为页（page）的块来查看的。在Pentium硬件上，默认页大小是4 096字节。物理内存（包括RAM和ROM）也都是分成页来查看的。当程序被加载到内存中执行时，操作系统会安排程序的部分页存储在物理内存的页中。这些页称为被“驻留”，其余部分存储在磁盘上。

在执行期间的各个阶段，将需要一些当前没有驻留的程序页。当发生这种情况时，硬件会感知到，将控制权转移给操作系统。后者将所需页带到内存中，可能会替换掉当前驻留的另一个程序页（如果没有可用的自由内存页），然后将控制权返回给程序。如果有被驱逐的程序页，就会变成非驻留页，被存储在磁盘上。

为了管理所有这些操作，操作系统为每个过程设立了一个*页表*（page table）。（Pentium的页表有一个层次结构，但是这里为了简单起见，我们假定只有一层，而且这里讨论的大多数内容都不是Pentium特有的。）这一过程的每个虚拟页在表中都有对应的一个项（entry），其中包括如下信息。

- 这个页在内存中或者磁盘上的当前物理位置。如果是在磁盘上，页表上对应的项会指示页是非驻留的，可能包含一个指针，指向最终导致磁盘上的物理位置的一个列表。例如，它可能显示：程序的虚拟页12是驻留的，位于内存的物理页200中。
- 该页的权限分3种：读、写和执行。

注意，操作系统不会将不完整的页分配给程序。例如，如果要运行的程序总共大约有10 000字节，如果完全加载，会占3个内存页。它不会仅占2.5个页，因为页是虚拟内存系统能够操作的最小内存单元。这是调试时要记住的很重要的一点，因为正如下面将介绍的，这一点暗示了程序的一些错误内存访问不会触发段错误。换言之，在调试会话期间，不能这么想：“这行代码一定没问题，因为它没有引起段错误。”

4.1.4　页表的作用

采用图4-1中的虚拟地址空间，继续假设页大小为4 096字节。然后虚拟页0包含虚拟地址空间

的第0~4 095字节，页1包含第4 096~8 191字节，以此类推。

前面提到，当我们运行程序时，操作系统创建一个用来管理执行程序代码进程的虚拟内存页表。（第5章介绍线程时将概述操作系统进程。）每当该进程运行时，硬件的页表寄存器都会指向该表。

从概念上讲，进程虚拟地址空间的每个页在页表中都有一个页表项（在实践中，可以使用各种技巧来压缩该表）。这个页表项存储与该页相关的各块信息。与段错误相关的数据是该页的访问权限，它类似于文件访问权限：读、写和执行。例如，页3的页表项将指出进程是否具有从该页读取数据的权限，向该页中写数据的权限，以及在该页上执行指令的权限（如果页中包含机器码）。

当程序执行时，它会如上文所述连续访问各个区域，导致硬件按以下几种情况处理页表。

- 每次程序使用其全局变量时，需要具有对数据区域的读/写访问权限。
- 每次程序访问局部变量时，程序会访问栈，需要对栈区域具有读/写访问权限。
- 每次程序进入或离开函数时，会对该栈进行一次或多次访问，需要对栈区域具有读写访问权限。
- 每次程序访问通过调用`malloc()`或`new`创建的存储空间时，都会发生堆访问，也需要读/写访问权限。
- 程序执行的每个机器指令都是从文本区域（或者从动态链接代码的区域）取出的，因此需要具有读和执行权限。

在程序的执行期间，生成的地址会是虚拟的。当程序试图访问某个虚拟地址处的内存时，比如y，硬件就会将其转换成虚拟页号v，它等于y除以4 096（其中除法是整除算法，舍去余数）。然后硬件会检查页表中的页表项v来查看该页的权限是否与要执行的操作匹配。如果匹配，硬件就会从这个表项中得到所需位置的实际物理页号，然后完成请求的内存操作。但是如果该表项显示请求的操作不具有恰当的权限，硬件就会执行内部中断。这会导致跳转到操作系统的错误处理例程。然后，操作系统一般会宣告一个内存访问违例，并停止程序的执行（即从进程表和内存中去掉程序）。

程序中的错误会导致权限不匹配，并在上面列出的某个类型的内存访问期间生成段错误。例如，假设程序中包含如下全局声明：

```
int x[100];
```

并假设代码包含下面这条语句：

```
x[i] = 3;
```

在C/C++中，表达式`x[i]`等价于（实际上也意味着）`*(x+i)`，即地址`x+i`指向的内存位置的内容。如果偏移量`i`为200 000，那么这个表达式可能会产生虚拟内存地址y，它超出了操作系统为该程序的数据区域指定的页组范围（编译器和连接程序为数组`x[]`安排的存储地方）。然后当试图执行写操作时会发生段错误。

如果x是局部变量，那么会在栈区域发生同样的问题。

与执行权限相关的违例的产生更微妙。例如，在汇编语言程序中，假设有一个名为sink的数据项和一个名为sunk()的函数。当调用这一函数时，可能不小心写成：

```
call sink
```

而不是

```
call sunk
```

这会导致一个段错误，因为程序会试图在sink的地址处执行指令，该地址位于数据区域，而数据区域的页没有启用执行权限。

C语言中类似这样的编码错误不会导致段错误，因为当将sink声明为变量时，编译器会以如下这行代码为目标：

```
z = sink(5);
```

但是在使用指向函数的指针时，很容易发生这种程序错误。来看如下的代码。

```
int f(int x)
{
   return x*x;
}
int (*p)(int);

int main( void )
{
   p = f;
   u = (*p)(5);
   printf("%d\n", u);

        return 0;
}
```

要是忘记了语句p = f;，那么p会为0，你将试图执行位于页0的指令，而你不具有对这个页的执行（或其他）权限（参见图4-1）。

4.1.5 轻微的内存访问程序错误可能不会导致段错误

为了加深对段错误发生方式的理解，来看如下代码，当执行该代码时，其行为表明：在预料到有段错误的地方不一定都会发生段错误。

```
int q[200];

main()
{
   int i;
```

```
    for (i = 0; i < 2000; i++) {
        q[i] = i;
    }
}
```

注意，这名程序员显然在循环中有笔误，设置了2 000次迭代，而不是200次。无论数组索引是否超出了边界，C语言编译器在编译时不会捕获这一错误，编译器生成的机器码在执行时也不会检查该错误。（这是GCC的错误，尽管它还具有不提供这样的运行时索引检查的`-fmudflap`选项。）

在执行时，很可能发生段错误。然而，错误发生的时机可能让人大吃一惊。这种错误可能不是出现在“自然”时间，即当`i`=200时；相反，它可能出现在那个时刻后面很久。

为了说明这一点，我们在Linux PC上的GDB下运行该程序，这样可以方便地查询变量的地址。结果发现段错误不是发生在`i`=200处，而是在`i`=728处。（你的系统可能会给出不同的结果，但原理是一样的。）让我们看一下原因。

通过对GDB的查询，我们发现数组`q[]`在地址`0x80497bf`处结束，即`q[199]`的最后一字节在该内存位置。考虑到Intel的页大小是4 096字节，这种机器是32位的字大小，所以一个虚拟地址被分解为一个20位的页号和一个12位的偏移量。在本节的示例中，`q[]`在虚拟页号`0x8049`=32841处结束，偏移量为`0x7bf`=1983。因此，分配了`q`的内存的页上仍然有4 096−1 984=2 112字节。该空间可以存放2 112/4=528个整数变量（因为这里使用的机器上每个变量是4字节宽），本节示例的代码似乎认为包含`q`的元素的位置是在200～727之间。

当然，`q[]`的那些元素不存在，但是编译器没有抱怨。硬件也没有抱怨，因为这种写操作仍然是在我们肯定具有写权限的页上执行的（因为`q[]`的部分实际元素位于该数据段上，且在其中分配了`q[]`）。只有当`i`变成728时，`q[i]`才引用不同页上的地址。在这种情况下，它就是我们不具有写权限（或任何其他权限）的页；虚拟内存硬件检测到这一点，并触发一个段错误。

由于每个整数变量都存储在4字节中，因此这个页包含528（2 112/4）个额外的“幻影”元素，代码将它作为属于数组`q[]`对待。因此，虽然我们没有打算让它这样处理，但是访问`q[200]`，`q[201]`…一直到元素199+518=727，即`q[727]`仍然是合法的，不会触发段错误！只有当试图访问`q[728]`时才会遇到新页，你可能有也可能没有对它的必需访问权限。这里，我们不具有对该页的访问权限，因此发生了段错误。然而，下一个页可能侥幸具有指定给它的恰当权限，然后还会有更多幻影数组元素。

总结：正如早先所介绍的，不能根据没有发生段错误来得出结论认为内存操作是正确的。

4.1.6　段错误与Unix信号

在上面的讨论中，我们说段错误一般会导致程序的中止。虽然这是正确的，但是想要认真彻底地调试，还应该了解有关Unix信号的内容。

信号（signal）表示在程序执行期间报告的异常情况，允许操作系统（或程序员自己的代码）反映多种事件。信号可能在某个进程上由系统的底层硬件抛出（`SIGSEGV`或`SIGFPE`），或者由操作

系统抛出（SIGTERM或SIGABRT），或者由另一个进程抛出（SIGUSR1或SIGUSR2），甚至可能由该进程本身发送（通过raise()库调用）。

最简单的信号示例为：当程序正在运行时按下键盘上的Ctrl+C组合键。按下（或释放）键盘上的任何键，都会生成导致操作系统例程运行的硬件中断。当按下Ctrl+C组合键时，操作系统认出这种键组合是一个特殊模式，并为控制终端上的进程抛出一个名为SIGINT的信号。通常的说法是操作系统“向进程发送一个信号”。我们将采用这种说法，但是重要的是要意识到实际上没有任何内容“发送”给进程。所发生的事情只不过是操作系统将信号记录到进程表中，以便下次进程接收信号时得到CPU上的时间片，执行恰当的信号处理程序，下面将会解释。（然而，假设有紧急信号，操作系统也可能会决定让接收进程的下一个时间片稍早于其他进程。）

一个进程上可以发出很多种不同的信号。在Linux中，可以通过在shell提示符后面键入如下代码来列出全部的信号。

```
man 7 signal
```

信号的定义有多种标准，比如POSIX.1，这些信号会出现在所有兼容的操作系统上。还有个别操作系统独有的信号。

4

每个信号都有自己的信号处理程序，这是当进程发出特定信号时调用的函数。回到我们的Ctrl+C示例中，当发现SIGINT时，操作系统将进程的当前指令设置在该特定信号的信号处理程序的开端。因此，当进程恢复时，它会执行该处理程序。

每种类型的信号有一个默认*信号处理程序*，除非实在需要，否则就不必亲自编写信号处理程序。默认情况下，忽略大多数无害信号。有些类型的信号较严重，比如违反内存访问权限产生的信号，表示出现了程序不能或不应继续执行的情况。在这样的情况下，默认信号处理程序会直接终止程序。

虽然有些信号处理程序不能被重写，但是在很多情况下可以编写自己的处理程序来替换操作系统提供的默认处理程序。在Unix下就可以用signal()或sigaction()两个系统调用。[①]例如，你的自定义处理程序函数可以忽略信号，甚至可以要求用户选择一个做法。

为了增加趣味性，我们编写了一个程序来说明如何编写自己的处理程序，并使用signal()调用或重写默认操作系统处理程序，或者忽略信号。我们选择了SIGNIT，但是可以对可被捕获的任何信号做相同的事情。该程序还演示了如何使用raise()。

```
#include <signal.h>
#include <stdio.h>

void my_sigint_handler( int signum )
{
```

① 有两个函数可用来重写默认信号处理程序，因为和其他Unix系统一样，Linux也遵循多个标准。signal()函数比sigaction()更容易使用，它遵循ANSI标准；而sigaction()函数较复杂，但是也更多样化，它遵循POSIX标准。

```
    printf("I received signal %d (that's 'SIGINT' to you).\n", signum);
    puts("Tee Hee!  That tickles!\n");
}


int main(void)
{
    char choicestr[20];
    int  choice;

    while ( 1 )
    {
        puts("1. Ignore control-C");
        puts("2. Custom handle control-C");
        puts("3. Use the default handler control-C");
        puts("4. Raise a SIGSEGV on myself.");
        printf("Enter your choice: ");

        fgets(choicestr, 20, stdin);
        sscanf(choicestr, "%d", &choice);

        if ( choice == 1 )
            signal(SIGINT, SIG_IGN);  // Ignore control-C
        else if ( choice == 2 )
            signal(SIGINT, my_sigint_handler);
        else if ( choice == 3 )
            signal(SIGINT, SIG_DFL);
        else if ( choice == 4 )
            raise(SIGSEGV);
        else
            puts("Whatever you say, guv'nor.\n\n");
    }

    return 0;
}
```

当程序违反内存访问权限时，在进程上发出SIGSEGV信号。默认段错误处理程序终止该进程，并向磁盘上写一个“核心文件”（很快会介绍）。

如果希望保持程序有效，而不是允许程序被终止，则可以为SIGSEGV编写一个自定义处理程序。事实上，有时可能要故意导致段错误，以便使某种工作得以完成。例如，有些并行处理软件包使用人为段错误，会有特殊处理程序响应这种错误以保持系统各个环节之间的一致性，4.3节将会介绍。第7章将介绍SIGSEGV的一些专用处理程序的另一类用途，涉及检测段错误并优雅地做出反应的一些工具。

然而，使用GDB/DDD/Eclipse时，自定义信号处理程序可能会使程序变复杂。无论是直接使用还是通过DDD GUI，每当发出任何信号时，GDB都会停止进程。在刚刚提到的类似于并行处理软件应用程序中，这意味着GDB会因为与调试工作无关的原因而非常频繁地停止。为了处理这种情况，需要使用`handle`命令告诉GDB在某些信号发生时不要停止。

4.1.7　其他类型的异常

除了段错误以外，还有一些其他原因也会导致崩溃。浮点异常（Floating-point exception，FPE）导致发出`SIGFPE`信号。虽然这个信号被称为“浮点”异常，但是它也包括整数算术异常，比如溢出和除以0的情况。在GNU和BSD系统上，传递给FPE处理程序的第二个参数指出了FPE的原因。默认处理程序在有些情况下（比如浮点溢出）将忽略`SIGFPE`，在另外一些情况下（比如整数除以0）则终止进程。

当CPU在执行机器指令期间在总线上检测到反常情况时，会发生总线错误。不同的架构对总线上发生的事情有不同的要求，这种反常的确切原因取决于具体的架构。导致总线错误的部分情况举例如下。

- 访问不存在的物理地址。这不同于段错误，段错误涉及访问不具备足够权限的内存。段错误是权限的问题，总线错误则是由于提供给处理器的是无效地址。
- 在很多架构上，要求访问32位量的机器指令要求字对齐，即这个量的内存地址必须是4的倍数。导致试图在奇数号地址上访问具有4字节的数的指针错误可能会引起总线错误。在x86架构上运行Linux时，下面这个程序不会引起总线错误，因为这些处理器未对齐的内存地址也是合法的，它们只是执行起来比对齐访问慢而已。

```
int main(void)
{
   char *char_ptr;
   int  *int_ptr;
   int   int_array[2];

   // char_ptr points to first array element
   char_ptr = (char *) int_array;

   // Causes int_ptr to point one byte past the start of an existing int.
   // Since an int can't be only one byte, int_ptr is no longer aligned.
   int_ptr = (int *) (char_ptr+1);

   *int_ptr = 1;                    // And this might cause a bus error.
   return 0;
}
```

无论哪种情况，总线错误是处理器层的异常，导致在Unix系统上发出`SIGBUS`信号。默认情况下，`SIGBUS`会导致进程转储核心并立即终止。

4.2　核心文件

正如前面所提到的，有些信号表示让某个进程继续是不妥当，甚至是不可能的。在这些情况中，默认动作是提前终止进程，并编写一个名为核心文件（core file）的文件，俗称转储核心。你的shell可能会禁止编写核心文件（参见4.2.2节了解详细信息）。

如果在程序运行期间创建了核心文件，则可以对该文件打开调试器（比如GDB），然后开始常规GDB操作。

4.2.1　核心文件的创建方式

核心文件包含程序崩溃时对程序状态的详细描述：栈的内容（或者，如果程序是多线程的，则是各个线程的栈），CPU寄存器的内容（同样，如果程序是多线程的，则是每个线程上的一组寄存器值），程序的静态分配变量的值（全局与`static`变量），等等。

创建核心文件非常容易。下面是生成这样文件的代码清单4-1。

代码清单4-1　*abort.c*

```
int main(void)
{
   abort();

   return 0;
}
```

`abort()`函数导致当前进程接收`SIGABRT`信号，`SIGABRT`的默认信号处理程序终止程序并转储核心。代码清单4-2是另一个转储核心的简短程序。在这个程序中，我们故意解除对NULL指针的引用。

代码清单4-2　*sigsegv.c*

```
int main(void)
{
   char *c = 0;
   printf("%s\n", *c);

   return 0;
}
```

让我们生成一个核心文件。编译并运行*sigsegv.c*。

```
$ gcc -g -W -Wall sigsegv.c -o sigsegv
$ ./sigsegv
Segmentation fault (core dumped)
```

如果列出当前目录，将注意到一个名为*core*（或者其变体）的新文件。当在文件系统中的某

处看到一个核心文件时，可能不能立即分辨出是哪个程序生成它的。Unix命令`file`有助于指出转储这个特定核心文件的可执行文件的名称。

```
$ file core
core: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style,
SVR4-style, from 'sigsegv'
```

核心文件命名约定

过去，核心文件的命名约定比较简单：它们都称为*core*。

后来，在GNU/Linux下，多线程程序开始用*core.3928*这样的文件名来转储核心，其中数字部分表示转储核心的进程ID。

从Linux 2.5内核起，可以使用*/proc/sys/kernel/*接口控制核心文件的名称。这个机制很简单，完好地用文档记录在Linux内核源树下的*Documentation/sysctl/kernel.txt*中。

4.2.2 某些shell可能禁止创建核心文件

很多情况下，调试过程都不涉及核心文件。如果程序发生了段错误，程序员只要打开调试器，比如GDB，并再次运行程序就可以重建该错误。由于这个原因，而且因为核心文件可能比较大，所以大多数现代shell都会在一开始就防止编写核心文件。

4

在`bash`中，可以使用`ulimit`命令控制核心文件的创建。

```
ulimit -c n
```

其中*n*是核心文件的最大大小，以千字节为单位。超过*n*KB的任何核心文件都不会被编写。如果没有指定*n*，shell就会显示核心文件上的当前限制。如果允许任意大小的核心文件，可以使用：

```
ulimit -c unlimited
```

对于`tcsh`和`csh`用户，`limit`命令控制核心文件大小。例如，

```
limit coredumpsize 1000000
```

告诉shell，如果核心文件大小大于1 000 000字节，则不创建该文件。

如果在运行sigsegv后没有得到核心文件，对于`bash`使用`ulimit -c`检查当前核心文件的限制，对于`tcsh`或`csh`则使用`limit -c`检查。

为什么需要首先有核心文件呢？既然可以简单地在GDB中重新运行具有段错误的程序，并重建段错误，为什么要那么麻烦地创建核心文件呢？答案是在有些情况下，比如在下面的情况下，这种假设是不成立的。

- 只有在运行了一段长时间后才发生段错误，所以在调试器中无法重新创建该错误。
- 程序的行为取决于随机的环境事件，因此再次运行程序可能不会再现段错误。
- 当新手用户运行程序时发生的段错误。这里的用户，通常不是程序员（也不具有对源代

码的访问权限），不会进行这种调试。然而，为了检查和调试，这样的用户可能仍然会向程序员发送核心文件（如果可以创建这种文件的话）。

然而要注意，如果程序的源代码不可用，或者可执行文件不是用增强的符号表编译的，甚或当我们没有计划调试可执行文件，核心文件可能都不会有太大的用处。

4.3 扩展示例

本节提供一个调试段错误的详细示例。

下面是一些C代码，可能是类似于C++字符串的托管字符串类型的实现部分。源文件*cstring.c*中包含的代码，实现一种称为CString的类型；然而，该代码充满了或明或暗的程序错误。我们的目标是找出所有这些程序错误并纠正它们。

CString是如下这种结构的别名：包含指向char字符串的存储空间的指针，以及一个字符串长度的变量。我们实现了一些适用于处理字符串的实用函数。

- **Init_CString()**：采用老式的C字符串作为一个实参，并用它来初始化新CString。
- **Delete_CString()**：**CStrings**是在堆上分配的，当不再需要时，必须释放它们的内存。这个函数负责无用单元收集。
- **Chomp()**：删除和返回CString的最后一个字符。
- **Append_Chars_To_CString()**：将C样式的字符串附加到CString后面。

最后，main()是测试CString实现的驱动函数。

本节的代码使用了一个极其有用的库函数snprintf()。万一你还没有遇到过这个函数，不明白它是什么，我简单解释如下：它基本上类似于printf()，只是这个函数将其输出写到字符数组中，而不是写到stdout中。为了帮助防止缓冲区溢出（在任何复制以空字符结尾的字符串的函数中，如果空字符被留在源字符串之外没有复制，则可能发生这种溢出），snprintf()还可以指定要写的最大字节数量，包括拖尾的空字符。

```
#include <stdio.h>
#define STRSIZE 22

int main(void)
{
   char s1[] = "brake";
   char *s2 = "breakpoints";
   char logo[STRSIZE];

   snprintf(logo, STRSIZE, "%c %s %d %s.", 'I', s1, 2+2, s2);

   puts(logo);
   return 0;
}
```

这个程序会将字符串“I brake 4 breakpoints”（我捣碎了4个断点）写到字符数组logo中，准备打印到一张保险杠贴纸上。

下面是我们的CString的实现。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
   char *str;
   int   len;
} CString;

CString *Init_CString(char *str)
{
   CString *p = malloc(sizeof(CString));
   p->len = strlen(str);
   strncpy(p->str, str, strlen(str) + 1);
   return p;
}


void Delete_CString(CString *p)
{
   free(p);
   free(p->str);
}


// Removes the last character of a CString and returns it.
//
char Chomp(CString *cstring)
{
   char lastchar = *( cstring->str + cstring->len);
   // Shorten the string by one
   *( cstring->str + cstring->len) = '0';
   cstring->len = strlen( cstring->str );

   return lastchar;
}


// Appends a char * to a CString
//
CString *Append_Chars_To_CString(CString *p, char *str)
```

```
{
   char *newstr = malloc(p->len + 1);
   p->len = p->len + strlen(str);
   // Create the new string to replace p->str
   snprintf(newstr, p->len, "%s%s", p->str, str);
   // Free old string and make CString point to the new string
   free(p->str);
   p->str = newstr;

   return p;
}
int main(void)
{
   CString *mystr;
   char c;

   mystr = Init_CString("Hello!");
   printf("Init:\n  str: `%s' len: %d\n", mystr->str, mystr->len);
   c = Chomp(mystr);
   printf("Chomp '%c':\n  str:`%s' len: %d\n", c, mystr->str, mystr->len);
   mystr = Append_Chars_To_CString(mystr, " world!");
   printf("Append:\n  str: `%s' len: %d\n", mystr->str, mystr->len);

   Delete_CString(mystr);

   return 0;
}
```

研究该代码并猜一下输出应是什么。然后编译并运行它。

```
$ gcc -g -W -Wall cstring.c -o cstring
$ ./cstring
Segmentation fault (core dumped)
```

噢，出错了！我们需要做的第一件事是找出这个段错误是在何处发生的。然后可以试着解释为什么会发生这样的段错误。

在继续讲述之前提一下，我们隔壁办公室的同事Milton也在尝试修复该程序中的错误。与我们不同的是，Milton不知道如何使用GDB，因此他打算打开Wordpad，在代码中到处插入对`printf()`的调用，并重新编译程序，试图指出段错误在何处发生。让我们看看我们对程序的调试是不是比Milton快。

Milton用Wordpad，我们则用GDB分析核心文件。

```
$ gdb cstring core
Core was generated by `cstring'.
```

```
Program terminated with signal 11, Segmentation fault.
#0  0x400a9295 in strncpy () from /lib/tls/libc.so.6

(gdb) backtrace
#0  0x400a9295 in strncpy () from /lib/tls/libc.so.6
#1  0x080484df in Init_CString (str=0x80487c5 "Hello!") at cstring.c:15
#2  0x080485e4 in main () at cstring.c:62
```

根据回溯输出，段错误发生在第15行的`Init_CString()`中，是在调用`strncpy()`期间发生的。甚至不需要仔细查看代码，我们已经知道很可能我们在第15行上向`strncpy()`传递了一个`NULL`指针。

这时，Milton仍然在犹豫应该在何处插入第一个对`printf()`的调用呢。

4.3.1 第一个程序错误

GDB指出，段错误发生在第15行的`Init_CString()`中，因此将当前帧改为调用`Init_CString()`的那一帧。

```
(gdb) frame 1
#1  0x080484df in Init_CString (str=0x80487c5 "Hello!") at cstring.c:15
15              strncpy(p->str, str, strlen(str) + 1);
```

我们将这样应用确认原则：通过查看传递给`strncpy()`的各个指针参数（即`str`、`p`和`p->str`），依次确认它们的值与我们预想的一样。首先输出`str`的值：

```
(gdb) print str
$1 = 0x80487c5 "Hello!"
```

由于`str`是指针，因此GDB给我们的值是十六进制地址`0x80487c5`。由于`str`是`char`的指针，因此是字符串的地址，GDB很够朋友地告诉我们字符串的值："Hello!"虽然从上面的回溯输出中可以看到这是很明显的，但是我们无论如何应检查一下。因此，`str`不是`NULL`，它指向一个有效字符串，到目前为止一切正常。

现在让我们将注意力转移到其他指针参数，`p`和`p->str`。

```
(gdb) print *p
$2 = {
  str = 0x0,
  len = 6
}
```

问题现在很明显：`p->str`，它也是指向字符串的指针，是`NULL`。所以段错误的原因就出来了：我们试图写入到内存中的位置0，而这个位置对于我们是禁区。

但是什么能导致`p->str`（该结构下`CString`中的字符串指针）为`NULL`？再看看代码。

```
(gdb) list Init_CString
5       typedef struct {
6               char *str;
7               int   len;
8       } CString;
9
10
11      CString *Init_CString(char *str)
12      {
13              CString *p = malloc(sizeof(CString));
14              p->len = strlen(str);
15              strncpy(p->str, str, strlen(str) + 1);
16              return p;
17      }
18
```

我们看到在发生段错误的代码行前面只有两行代码，它们之间的第13行很可能是问题根源所在。

我们将从GDB中重新运行程序，在进入`Init_CString()`的入口处设置一个临时断点，并逐行单步调试这个函数，查看`p->str`的值。

```
(gdb) tbreak Init_CString
Breakpoint 1 at 0x804849b: file cstring.c, line 13.
(gdb) run

Breakpoint 1, Init_CString (str=0x80487c5 "Hello!") at cstring.c:13
13              CString *p = malloc(sizeof(CString));
(gdb) step
14              p->len = strlen(str);
(gdb) print p->str
$4 = 0x0
(gdb) step
15              strncpy(p->str, str, strlen(str) + 1);
```

问题在这里：我们打算提交一个段错误，因为下一行代码解除对`p->str`的引用，`p->str`仍然为`NULL`。现在我们使用小灰单元格来判断发生了什么。

当为`p`分配内存时，我们获得了存放`struct`的足够内存：一个存放字符串地址的指针，以及一个存放字符串长度的`int`变量。但是我们没有分配存放字符串本身的内存。我们犯了个常见的错误：声明了指针，却没有声明将指针指向任何对象！我们需要做的事首先是分配足够的内存来存放`str`，然后使`p->str`指向刚刚分配的内存。下面是这件事的做法（我们需要在字符串的长度上加1，因为`strlen()`没有将末尾的`'\0'`统计在内）。

```
CString *Init_CString(char *str)
{
```

```
    // Allocate for the struct
    CString *p = malloc(sizeof(CString));
    p->len = strlen(str);
    // Allocate for the string
    p->str = malloc(p->len + 1);
    strncpy(p->str, str, strlen(str) + 1);
    return p;
}
```

顺便说一下，Milton刚刚把所有`printf()`调用加进他的代码中，正打算重新编译。如果他幸运的话，就会发现在何处发生了段错误。如果不那么幸运，他将不得不添加更多`printf()`调用。

4.3.2　在调试会话期间不要退出GDB

在调试会话期间，修改代码时永远不要退出GDB。前面说过，这样就不必费时间来启动，可以保留我们的断点，等等。

类似地，我们要保持文本编辑器打开。在调试时的两次编译之间留在同一个编辑器会话中，我们可以充分利用编辑器的"撤销"功能。例如，调试过程中的一个常见策略是临时删除部分代码，以便集中精力研究你认为存在错误的余下部分。完成检查后，只要使用编辑器的撤销功能恢复被删除的代码行即可。

因此，在屏幕上我们通常有一个GDB（或DDD）窗口，以及一个编辑器窗口。我们还打开了第三个窗口用于执行编译器命令，甚至最好是通过编辑器执行命令。例如，如果使用的是Vim编辑器，可以执行如下命令，它会保存所做的编辑修改，并在同时重新编译程序。

```
: make
```

（我们还假定在Vim启动文件中使用`set autowrite`设置了Vim的`autowrite`变量。如果有错，Vim的功能也会将光标移到报告的第一个编译警告或错误处，可以通过Vim的`:cnext`和`:cprev`命令在多个编译错误中来回调试。当然，如果使用这些命令的简短别名放到Vim启动文件中，所有这些操作将更方便。）

4.3.3　第二个和第三个程序错误

当修复了第一个程序错误后，再次从GDB中运行程序。（记住，当GDB注意到重新编译了程序后，它会自动加载新的可执行文件，因此同样不需要退出和重启GDB。）

```
(gdb) run
     The program being debugged has been started already.
     Start it from the beginning? (y or n) y
`cstring' has changed; re-reading symbols.

Starting program: cstring

Init:
```

```
  str: `Hello!' len: 6
Chomp '':
  str:`Hello!0' len: 7
Append:
  str: `Hello!0 world' len: 14

Program exited normally.
(gdb)
```

Chomp()中似乎有两个问题。第一，它应加上一个感叹号'!'，但是看上去它加上的是非打印字符。第二，0字符出现在我们的字符串末尾。由于Chomp()是查找这些程序错误的明显地方，因此我们将启动该程序，并在Chomp()的入口处放置一个临时断点。

```
(gdb) tbreak Chomp
Breakpoint 2 at 0x8048523: file cstring.c, line 32.
(gdb) run
Starting program: cstring

Init:
  str: `Hello!' len: 6

Breakpoint 1, Chomp (cstring=0x804a008) at cstring.c:32
32              char lastchar = *( cstring->str + cstring->len);
(gdb)
```

该字符串的最后一个字符应该是'!'。让我们确认一下。

```
(gdb) print lastchar
$1 = 0 '\0'
```

我们预期lastchar是'!'，但实际上它是一个空字符。这看上去可能是一个“差一”（off by one）错误。让我们来找出这个问题。可以按如下代码所示可视化下面的字符串。

```
pointer offset: 0 1 2 3 4 5 6
cstring->str:   H e l l o ! \0
string length:  1 2 3 4 5 6
```

这个字符串的最后一个字符存储在地址cstring->str+5上，但是因为该字符串长度是字符计数，而不是索引，因此地址cstring->str + cstring->len指向最后一个字符后面的一个数组位置，在这个位置上是以NULL结束，而不是我们希望它指向的位置。这个问题可以修复，方法是将如下代码：

```
char lastchar = *( cstring->str + cstring->len);
```

改为：

```
char lastchar = *( cstring->str + cstring->len - 1);
```

这部分代码中隐藏了第三个程序错误。当调用Chomp()后，字符串“Hello!”变成“Hello!0”（不是“Hello”）。在GDB中要执行的下一行（第33行）是要缩短字符串的地方，方法是使用结尾的空字符替换其最后一个字符。

```
*( cstring->str + cstring->len) = '0';
```

我们很快发现这一行里包含了我们刚刚在第31行修复过的同样问题：错误地引用了字符串的最后一个字符。而且，由于我们的眼睛已经适应这行代码，因此似乎我们在该位置存储字符'0'，它不是空字符。我们的意思是将'\0'放在字符串的末尾。进行了这两处纠正后，第33行为：

```
*( cstring->str + cstring->len - 1) = '\0';
```

这时，我们的同事Milton使用printf()找到了第一个段错误，并且刚刚在Init_CString()中纠正了内存分配问题。他没有继续前进找到我们刚刚在Chomp()中修复的程序错误，而是不得不将所有对printf()的调用都去掉，并重新编译程序。他真累！

4.3.4　第四个程序错误

根据上一节的讨论我们进行一些纠正，重新编译代码，并再次运行程序。

```
(gdb) run
`cstring' has changed; re-reading symbols.
Starting program: cstring

Init:
  str: `Hello!' len: 6
Chomp '!':
  str:`Hello' len: 5
Append:
  str: `Hello world' len: 12
Program received signal SIGSEGV, Segmentation fault.
0xb7f08da1 in free () from /lib/tls/libc.so.6
(gdb)
```

又出现一个段错误。根据附加操作后缺少惊叹号来判断，下一个程序错误有可能隐藏在Append_Chars_To_CString()中。可以使用一个简单的backtrace来确认或否认这种假设。

```
1  (gdb) backtrace
2  #0  0xb7f08da1 in free () from /lib/tls/libc.so.6
3  #1  0x0804851a in Delete_CString (p=0x804a008) at cstring3.c:24
4  #2  0x08048691 in main () at cstring3.c:70
5  (gdb)
```

根据回溯输出的第三行发现，我们的假设是错误的：程序实际上在Delete_CString()中崩溃。这并不意味着我们在Append_Chars_To_CString()中也没有程序错误，只是导致段错误的直接相

关程序错误在`Delete_CString()`中。这正是这里使用GDB检查我们预期的原因——完全省略了查找在何处发生段错误的工作。一旦我们那位使用`printf()`的朋友在调试时赶到这里，他一定会将追踪代码放在错误的函数中！

幸而，`Delete_CString()`较短，因此应该能很快发现错误所在。

```
(gdb) list Delete_CString
20
21    void Delete_CString(CString *p)
22    {
23        free(p);
24        free(p->str);
25    }
26
```

我们首先释放`p`，然后释放`p->str`。这是一个不太难发现的错误。当释放`p`后，不能保证`p->str`还能指向内存中的正确位置，它可能指向任何位置。当这个“任何位置”是我们不能访问的内存时，就会引起段错误。修复方法是逆转对`free()`的调用顺序。

```
void Delete_CString(CString *p)
{
    free(p->str);
    free(p);
}
```

顺便说一下，Milton在尝试跟踪我们轻易修复的`Chomp()`中的偏移错误时屡战屡败。所以他打电话向我们求助。

4.3.5　第五个和第六个程序错误

我们纠正、重新编译，并再次运行代码。

```
(gdb) run
`cstring' has changed; re-reading symbols.
Starting program: cstring

Init:
  str: `Hello!' len: 6
Chomp '!':
  str:`Hello' len: 5
Append:
  str: `Hello world' len: 12

Program exited normally.
(gdb)
```

在附加操作以后，字符串中丢失了惊叹号，本来应当是"Hello world!"。奇怪的是，尽管字符串是错误的，但报告的字符串长度12却是正确的。查找这个程序错误的最符合逻辑的位置在Append_Chars_To_CString()中，因此在那里放一个断点。

```
(gdb) tbreak Append_Chars_To_CString
Breakpoint 3 at 0x8048569: file cstring.c, line 45.
(gdb) run
Starting program: cstring

Init:
  str: `Hello!' len: 6
Chomp '!':
  str:`Hello' len: 5

Breakpoint 1, Append_Chars_To_CString (p=0x804a008, str=0x8048840 " world!")
    at cstring.c:45
45              char *newstr = malloc(p->len + 1);
```

C字符串newstr需要足够大，以便同时存放p->str和str。我们看到，在第45行对malloc()的调用没有分配足够的内存，它仅仅分配了够放p->str和一个结尾空字符的空间。第45行应改为：

```
char *newstr = malloc(p->len + strlen(str) + 1);
```

当进行了这种纠正并重新编译后，得到如下输出。

```
(gdb) run
`cstring' has changed; re-reading symbols.
Starting program: cstring

Init:
  str: `Hello!' len: 6
Chomp '!':
  str:`Hello' len: 5
Append:
  str: `Hello world' len: 12
```

这一纠正没有修复我们惦记着的程序错误。我们发现并修复的是一个“静默的程序错误”。没错：它确实是一个程序错误，它没有作为段错误出现，纯粹是一种运气。其余程序错误很有可能仍然在Append_Chars_To_CString()中，因此我们在那里设置另一个断点。

```
(gdb) tbreak Append_Chars_To_CString
Breakpoint 4 at 0x8048569: file cstring.c, line 45.
(gdb) run
Starting program: cstring
```

```
(gdb) run
Init:
  str: `Hello!' len: 6
Chomp '!':
  str:`Hello' len: 5

Breakpoint 1, Append_Chars_To_CString (p=0x804a008, str=0x8048840 " world!")
    at cstring.c:45
45              char *newstr = malloc(p->len + strlen(str) + 1);
(gdb) step
46              p->len = p->len + strlen(str);
```

第46行显示了为什么尽管字符串本身是错误的，字符串长度却是正确的：加号正确地计算了`p->str`与`str`连接起来的长度。这里没有问题了，因此可以继续下一步。

```
(gdb) step
49              snprintf(newstr, p->len, "%s%s", p->str, str);
```

下一行代码（第49行）是我们形成新字符串的地方。我们预期在这一步后`newstr`会包含`"Hello world!"`。让我们应用确认原则来验证这一点。

```
(gdb) step
51              free(p->str);
(gdb) print newstr
$2 = 0x804a028 "Hello world"
```

第51行上代码构造的字符串中缺少了惊叹号，因此程序错误可能发生在第49行，但会是什么错误呢？在对`snprintf()`的调用中，我们请求至多将`p->len`字节都复制到`newstr`中。`p->len`的值被确认为12，下一个`"Hello world!"`有12个字符。我们没有让`snprintf()`复制原字符串中的结尾空字符。然而，本来以为会得到一个畸型字符串，最后一个位置有惊叹号，没有了空字符，但实际上并非如此。

这是`snprintf()`的出色之处。它总是将结尾空字符复制到目标字符串中。如果你偷懒，指定复制的最多字符数小于原字符串中实际字符数（正如我们这里所做的），`snprintf()`就会复制它能够复制的尽可能多的字符，但写入目标字符串的最后一个字符肯定是空字符。为了修复我们的错误，需要让`snprintf()`复制足够的字节以存放源字符串的文本和结尾空字符。

因此需要修改第45行。下面是完整的修复后函数。

```
CString *Append_Chars_To_CString(CString *p, char *str)
{
   char *newstr = malloc(p->len + strlen(str) + 1);
   p->len = p->len + strlen(str);

   // Create the new string to replace p->str
```

```
    snprintf(newstr, p->len + 1, "%s%s", p->str, str);
    // Free old string and make CString point to the new string
    free(p->str);
    p->str = newstr;

    return p;
}
```

让我们重新编译修复后的代码，并运行程序。

```
(gdb) run
`cstring' has changed; re-reading symbols.
Starting program: cstring

Init:
  str: `Hello!' len: 6
Chomp '!':
  str:`Hello' len: 5
Append:
  str: `Hello world!' len: 12

Program exited normally.
(gdb)
```

看上去不错！

虽然我们涵盖了很多领域，并遇到了一些比较难的概念，但这是值得的。即使我们的含程序错误的`CString`实现有点做作，但我们的调试会话是相当实用的，包括了调试的很多方面：

- 确认原则；
- 使用核心文件进行崩溃进程的“死后”分析；
- 纠正、编译并重新运行程序，甚至不需要退出GDB；
- `printf()`风格调试的不足之处；
- 利用你的智慧，这虽然原始，却是无可替代的。

如果你过去是用`printf()`风格调试的，就会发现使用`printf()`跟踪某些程序错误会有多难。虽然在调试中不可避免地会使用`printf()`诊断代码，但是作为一种通用目的的“工具”，它远远不足以用来跟踪实际代码中发生的大部分程序错误。

第 5 章

多活动上下文中的调试

调试本来就不是件容易事，当有问题的应用程序试图协调多个同步活动时，调试就更加具有挑战性。客户/服务器网络编程、多线程编程，以及并行处理，都属于这种情况。本章将概述最常用的多路编程技术，并提供一些方法和技巧处理这些类型的程序中的错误，着重介绍调试过程中GDB/DDD/Eclipse的用法。

5.1 调试客户/服务器网络程序

计算机网络是极其复杂的系统，网络化软件应用程序的精确调试有时需要使用硬件监控来收集关于网络通信量的详细信息。单就这种调试主题，就足以写一本书。我们这里的目标只是简单地介绍一下这一主题。

我们的示例由下面的客户/服务器对组成。用户可以通过客户机应用程序检查运行服务器应用程序机器上的负荷，即使用户没有服务器所在机器的账户也可以。客户机通过网络连接向服务器发送查询信息的请求（这里是通过Unix的w命令查询服务器所在系统上的负荷）。然后服务器处理该请求并返回结果，捕获w的输出，并将它通过网络连接发送回客户机。一般来说，一个服务器可以接受来自多个远程客户机的请求，为了让示例简单一点，我们假设只有一个客户机。

服务器的代码如下所示。

```
1  // srvr.c
2
3  //  a server to remotely run the w command
4  //  user can check load on machine without login privileges
5  //  usage:  svr
6
7  #include <stdio.h>
8  #include <sys/types.h>
9  #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12 #include <fcntl.h>
13 #include <string.h>
```

```
14  #include <unistd.h>
15  #include <stdlib.h>
16
17  #define WPORT 2000
18  #define BUFSIZE 1000  // assumed sufficient here
19
20  int clntdesc,  // socket descriptor for individual client
21      svrdesc;  // general socket descriptor for server
22
23  char outbuf[BUFSIZE];  // messages to client
24
25  void respond()
26  {  int fd,nb;
27
28     memset(outbuf,0,sizeof(outbuf));  // clear buffer
29     system("w > tmp.client");  // run 'w' and save results
30     fd = open("tmp.client",O_RDONLY);
31     nb = read(fd,outbuf,BUFSIZE);  // read the entire file
32     write(clntdesc,outbuf,nb);  // write it to the client
33     unlink("tmp.client");  // remove the file
34     close(clntdesc);
35  }
36
37  int main()
38  {  struct sockaddr_in bindinfo;
39
40     // create socket to be used to accept connections
41     svrdesc = socket(AF_INET,SOCK_STREAM,0);
42     bindinfo.sin_family = AF_INET;
43     bindinfo.sin_port = WPORT;
44     bindinfo.sin_addr.s_addr = INADDR_ANY;
45     bind(svrdesc,(struct sockaddr *) &bindinfo,sizeof(bindinfo));
46
47     // OK, listen in loop for client calls
48     listen(svrdesc,5);
49
50     while (1)  {
51        // wait for a call
52        clntdesc = accept(svrdesc,0,0);
53        // process the command
54        respond();
55     }
56  }
```

下面是客户机的代码。

```
// clnt.c

// usage:  clnt server_machine

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>

#define WPORT 2000  // server port number
#define BUFSIZE 1000

int main(int argc, char **argv)
{  int sd,msgsize;

   struct sockaddr_in addr;
   struct hostent *hostptr;
   char buf[BUFSIZE];

   // create socket
   sd = socket(AF_INET,SOCK_STREAM,0);
   addr.sin_family = AF_INET;
   addr.sin_port = WPORT;
   hostptr = gethostbyname(argv[1]);
   memcpy(&addr.sin_addr.s_addr,hostptr->h_addr_list[0],hostptr->h_length);

   // OK, now connect
   connect(sd,(struct sockaddr *) &addr,sizeof(addr));

   // read and display response
   msgsize = read(sd,buf,BUFSIZE);
   if (msgsize > 0)
      write(1,buf,msgsize);
   printf("\n");
   return 0;
}
```

对于不熟悉客户/服务器编程的读者，下面概述这种程序的工作原理。

服务器代码的第41行上创建了一个套接字（socket），这是一种类似于文件描述符的抽象，正如人们使用文件描述符在文件系统对象上执行I/O操作一样，人们通过套接字向网络连接读写信息。在第45行上，套接字与特定的端口号绑定，随机地选择为2000号端口。（像本例这样的用户

级别的应用程序只能使用1024以上的端口号。）这个数字标识了服务器所在系统上的“邮箱”，客户机要让这个特定应用程序处理的请求发送到该邮箱。

在第48行上，服务器通过调用listen()来“开张营业”。然后在第52行上通过调用accept()等待客户机请求进入。在接收到一个请求之前，该调用会阻塞。然后它返回一个新的套接字来与客户机通信。（当有多个客户机时，原来的套接字继续接收新请求，甚至有可能同时服务现有请求，因此需要单独的套接字。所以需要按多线程风格实现服务器。）服务器使用respond()函数处理客户机请求，通过本地调用w命令来将机器负荷信息发送给客户机，并在第32行将结果写到套接字中。

在第24行上，客户机创建一个套接字，然后在第31行使用这个套接字与服务器的端口2000连接。在第34行上，读取服务器发送的负荷信息，然后显示输出。

客户机的输出如下所示。

```
$ clnt laura.cs.ucdavis.edu
 13:00:15 up 13 days, 39 min,  7 users,  load average: 0.25, 0.13, 0.09
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
matloff  :0       -                14Jun07 ?xdm?  25:38   0.15s -/bin/tcsh -c /
matloff  pts/1    :0.0             14Jun07 17:34   0.46s  0.46s -csh
matloff  pts/2    :0.0             14Jun07 18:12   0.39s  0.39s -csh
matloff  pts/3    :0.0             14Jun07 58.00s  2.18s  2.01s /usr/bin/mutt
matloff  pts/4    :0.0             14Jun07  0.00s  1.85s  0.00s clnt laura.cs.u
matloff  pts/5    :0.0             14Jun07 20.00s  1.88s  0.02s script
matloff  pts/7    :0.0             19Jun07  4days 22:17   0.16s -csh
```

现在假设程序员忘记在客户机代码中编写第26行，这一行指定服务器的系统要连接到的端口。

```
addr.sin_port = WPORT;
```

让我们假装不知道程序错误是什么，看看如何跟踪到这个错误。

客户机的输出现在是：

```
$ clnt laura.cs.ucdavis.edu

$
```

似乎客户机从服务器什么也没有收到。这当然可能是由服务器或客户机上的各种原因引起的，或者服务器和客户机上都有原因。

让我们使用GDB检查一下。首先，确认客户机成功地连接到了服务器。在调用connect()时设置一个断点，并运行程序。

```
(gdb) b 31
Breakpoint 1 at 0x8048502: file clnt.c, line 31.
(gdb) r laura.cs.ucdavis.edu
Starting program: /fandrhome/matloff/public_html/matloff/public_html/Debug
```

5

```
/Book/DDD/clnt laura.cs.ucdavis.edu

Breakpoint 1, main (argc=2, argv=0xbf81a344) at clnt.c:31
31          connect(sd,(struct sockaddr *) &addr,sizeof(addr));
```

使用GDB执行connect()，并检查一个错误条件的返回值。

```
(gdb) p connect(sd,&addr,sizeof(addr))
$1 = -1
```

返回值真的是表示失败的代码-1。这是一个重要线索。（当然，这是防御性编程的问题，当编写客户机代码时，会检查connect()的返回值，并处理连接失败的情况。）

顺便说一下，注意在手工执行对connect()的调用时，必须去掉这种类型强制转换，如果保留了这样的类型强制转换，将会得到下面的错误消息。

```
(gdb) p connect(sd,(struct sockaddr *) &addr,sizeof(addr))
No struct type named sockaddr.
```

这是由于GDB中存在的一个怪异现象，因为我们在程序其他地方没有使用过这个结构，所以抛出了这条错误。

还要注意，如果GDB会话中的connect()成功了，就不能继续前进并执行第31行。试图打开已经打开的套接字是一个错误。

刚才可以跳过第31行，直接来到第34行。可以使用GDB的jump命令做这件事，执行jump 34，但是一般而言应当慎用这个命令，因为它可能导致跳过一些代码下一步需要的机器指令。因此，如果连接尝试成功，最好重新运行程序。

下面通过检查调用connect()时的实参addr来尝试跟踪失败的原因。

```
(gdb) p addr
...
connect(3, {sa_family=AF_INET, sin_port=htons(1032),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 ECONNREFUSED (Connection refused)
...
```

从上面的代码可以看出，值htons(1032)指示端口2052（见下面），而不是我们预期的2000。这说明有可能错误地指定了端口，或者是根本忘记了指定端口。检查一下，很快会发现是后一种情况。

再次，在源代码中增加帮助调试过程的机制（比如检查系统调用的返回值）时要谨慎。另一个有用的步骤是包括如下代码行：

```
#include <errno.h>
```

在我们的系统上，这行代码创建了一个全局变量errno，它的值可能从代码中或者GDB中输出。

```
(gdb) p errno
$1 = 111
```

对照文件*/usr/include/linux/errno.h*，发现这个错误码表示“连接被拒绝”错误。

然而，errno库的实现可能根据平台的不同而不同。例如，头文件可能有不同的名称，或者errno可能被作为宏调用实现，而不是作为变量实现。

另一个方法是使用strace，跟踪程序做过的所有系统调用。

```
$ strace clnt laura.cs
...
connect(3, {sa_family=AF_INET, sin_port=htons(1032),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 ECONNREFUSED (Connection refused)
...
```

真是一举两得。得到的第一个重要信息是，立即看到有一个ECONNREFUSED错误。第二个信息是，还看到端口为htons(1032)，其值是2052。通过从GDB中执行类似如下的命令可以检查后者的值：

```
(gdb) p htons(1032)
```

显示值为2052，显然不是预期的2000。

在很多情况下（无论是否与网络连接），strace都是检查系统调用结果的便捷工具。

再举一例，假设不小心忘记了在服务器代码中写入客户机（第32行）。

```
write(clntdesc,outbuf,nb);  // write it to the client
```

在这种情况下，客户机程序会挂起，等待没有得到的响应。当然，在这个简单示例中，很快可以想到是在服务器中调用write()时出了问题，因为忘记了在代码中写入客户机。但是在比较复杂的程序中，原因可能不会这么明显。在这样的情况下，需要建立两个同步GDB会话，一个用于客户机，一个用于服务器，一前一后地单步调试程序的这两个会话。然后可以发现，客户机在连接操作的某个位置挂起，等待服务器的响应，因而可以推断出程序错误可能在服务器中某个位置。然后将注意力集中在服务器GDB会话上，尝试弄清楚它在那时为什么没有向客户机发送响应。

在真正复杂的网络调试情况中，可以使用开源Ethereal程序跟踪单个TCP/IP分组。

5.2　调试多线程代码

目前多线程编程变得很流行。对于Unix，最普遍的线程包是POSIX标准Pthreads，因此本节将它用于我们的示例中。其他线程包的原理与它相似。

5.2.1　进程与线程回顾

现代操作系统使用分时技术管理多个运行程序，对用户来说似乎是同步执行的。当然，如果机器上有多个CPU，会有多个程序真正同步运行。但是为了简单起见，我们假设只有一个处理器，在这种情况下同步只是一种表面现象。

操作系统将运行程序的每个实例表述为进程（Unix术语）或任务（Windows术语）。因此，

同时执行的单个程序的多个调用（例如vi文本编辑器的同步会话）是各自独立的进程。在只有一个CPU的机器上，进程必须"轮换"操作。为了形象地说明，让我们假定每个"周期"（称为时间片）的长度为30毫秒。

当一个进程运行了30毫秒以后，硬件计时器发出一个中断，导致操作系统运行。我们假定这个进程被别的进程抢占了时间片。操作系统保存被中断进程的当前状态，因此以后可以恢复其状态，然后选择可以得到时间片的下一个进程。这个过程称为*环境切换*，因为CPU的执行环境从一个进程切换到另一个进程。这个循环无限重复。

轮换状态可能提前终结。例如，当进程需要执行输入/输出时，最终会调用操作系统中的一个执行低层硬件操作的函数，比如说调用C库函数`scanf()`会导致进行Unix OS `read()`系统调用，它需要与键盘驱动程序进行交互。这样，进程将它的轮换安排让给操作系统，提前结束了轮换状态。

这种情况说明了一个问题，给定进程时间片的调度是相当随机的。用户思考然后按下一个键所花的时间也是随机的，因此无法预测下次时间片何时开始。而且，如果调试的是多线程程序，则不知道将要调度的线程次序，这会使调试更加困难。

更细致地看：操作系统中有一个*进程表*，列出了关于当前所有进程的信息。一般来说，每个进程在进程表中被标记为Run状态或Sleep状态。让我们来看一个示例，其中一个运行程序到达需要从键盘读取输入的位置。正如刚刚提到的，这种情况会结束进程的轮换状态。因为进程现在在等待I/O完成，所以操作系统将其标记为处于Sleep状态，使它无资格占用时间片。因此，在Sleep状态意味着进程被阻塞，等待某个事件的发生。当这个事件最终发生后，操作系统会将它在进程表中的状态改回到Run。

非I/O事件也可以触发进程变成Sleep状态。例如，如果一个父进程创建了一个子进程，并调用`wait()`，那么父进程会阻塞，直到子进程完成它的工作并且结束。同样，这件事发生的确切时间是无法预测的。

而且，在Run状态中并不表示进程实际在CPU上执行；相反，这仅意味着它准备运行，即有资格获得处理器的时间片。一旦发生了环境切换，操作系统就根据进程表从当前处于Run状态的进程中选择一个进程占用CPU的一个周期。操作系统使用这一调度过程来选择新的环境，保证任何给定进程都能获得时间片，因而最终会完成，但是不承诺它会收到哪些时间片。因此，等待的事件发生后，睡眠进程真正"醒来"的确切时间是随机的，进程完成的准确速率也是随机的。

线程与进程非常类似，只是线程占用的内存比进程少，创建线程和在两个线程之间切换所需的时间也较少。事实上，线程有时被称为"轻量级"进程，根据线程系统和运行时环境，它们甚至可能被作为操作系统的进程实现。像使用进程完成工作的程序一样，多线程应用程序一般会执行一个`main()`过程，该过程创建一个或多个子线程。父线程`main()`也是线程。

进程和线程之间的主要区别是：与进程一样，虽然每个线程有自己的局部变量，但是多线程环境中父程序的全局变量被所有线程共享，并作为在线程之间通信的主要方法。（虽然也可以在Unix进程之间共享全局变量，但是这样做不方便。）

在Linux系统上，可以通过运行命令`ps axH`来查看系统上当前的所有进程和线程。

虽然有非抢占线程系统，但是Pthreads使用的是抢占线程管理策略，程序中的一个线程可能在任何时候被另一个线程中断。因此，上面描述的进程在时间共享系统中的随机性要素，同样出现在多线程程序的行为中。所以，使用Pthreads开发的应用程序中有些程序错误不太容易重现。

5.2.2 基本示例

下面介绍一个求素数的简单示例，该程序使用经典的埃拉托色尼筛法。要求2～n之间的素数，首先列出所有这些数字，然后去掉所有2的倍数，再去掉所有3的倍数，以此类推。最后剩下的就是素数。

```
1  // finds the primes between 2 and n; uses the Sieve of Eratosthenes,
2  // deleting all multiples of 2, all multiples of 3, all multiples of 5,
3  // etc.; not efficient, e.g. each thread should do deleting for a whole
4  // block of values of base before going to nextbase for more
5
6  // usage:  sieve nthreads n
7  // where nthreads is the number of worker threads
8
9  #include <stdio.h>
10 #include <math.h>
11 #include <pthread.h>
12
13 #define MAX_N 100000000
14 #define MAX_THREADS 100
15
16 // shared variables
17 int nthreads,  // number of threads (not counting main())
18     n,  // upper bound of range in which to find primes
19     prime[MAX_N+1],  // in the end, prime[i] = 1 if i prime, else 0
20     nextbase;  // next sieve multiplier to be used
21
22 int work[MAX_THREADS];  // to measure how much work each thread does,
23                         // in terms of number of sieve multipliers checked
24
25 // lock index for the shared variable nextbase
26 pthread_mutex_t nextbaselock = PTHREAD_MUTEX_INITIALIZER;
27
28 // ID structs for the threads
29 pthread_t id[MAX_THREADS];
30
31 // "crosses out" all multiples of k, from k*k on
32 void crossout(int k)
33 {  int i;
34
35    for (i = k; i*k <= n; i++)  {
```

5

```
36          prime[i*k] = 0;
37       }
38    }
39
40    // worker thread routine
41    void *worker(int tn)  // tn is the thread number (0,1,...)
42    {  int lim,base;
43
44       // no need to check multipliers bigger than sqrt(n)
45       lim = sqrt(n);
46
47       do  {
48          // get next sieve multiplier, avoiding duplication across threads
49          pthread_mutex_lock(&nextbaselock);
50          base = nextbase += 2;
51          pthread_mutex_unlock(&nextbaselock);
52          if (base <= lim)  {
53             work[tn]++;  // log work done by this thread
54             // don't bother with crossing out if base is known to be
55             // composite
56             if (prime[base])
57                crossout(base);
58          }
59          else return;
60       } while (1);
61    }
62
63    main(int argc, char **argv)
64    {  int nprimes,  // number of primes found
65           totwork,  // number of base values checked
66           i;
67       void *p;
68
69       n = atoi(argv[1]);
70       nthreads = atoi(argv[2]);
71       for (i = 2; i <= n; i++)
72          prime[i] = 1;
73       crossout(2);
74       nextbase = 1;
75       // get threads started
76       for (i = 0; i < nthreads; i++)  {
77          pthread_create(&id[i],NULL,(void *) worker,(void *) i);
78       }
79
80       // wait for all done
81       totwork = 0;
```

```
82      for (i = 0; i < nthreads; i++)  {
83         pthread_join(id[i],&p);
84         printf("%d values of base done\n",work[i]);
85         totwork += work[i];
86      }
87      printf("%d total values of base done\n",totwork);
88
89      // report results
90      nprimes = 0;
91      for (i = 2; i <= n; i++)
92         if (prime[i]) nprimes++;
93      printf("the number of primes found was %d\n",nprimes);
94
95   }
```

这个程序中有两个命令行参数，一个参数用来检查素数范围上边界的n，另一个参数是表示要创建的工作线程数量的nthreads。

这里的main()创建工作线程，每个工作线程是对函数worker()的一次调用。这些工作线程共享3个数据项：上边界变量n，指定要从2～n的范围内消除其倍数的下一个数字的变量nextbase，以及记录2～n的范围内的每个数字是否被消除的数组prime[]。每次调用反复地取得一个尚未处理的被乘数base，然后消除2～n范围内base的所有倍数。创建了工作线程后，main()就使用pthread_join()等待所有线程完成各自的工作，然后在统计留下的素数的数量后恢复程序，并发出报告。报告中不仅包括素数的数量，而且包括每个工作线程完成了多少工作的信息。在多处理器系统上进行负载平衡和性能优化时这种评估很有用。

worker()的每个实例通过执行如下代码取得base的下一个值。

```
pthread_mutex_lock(&nextbaselock);
base = nextbase += 2;
pthread_mutex_unlock(&nextbaselock);
```

这里更新了全局变量nextbase，用它来初始化worker()实例的局部变量base的值；然后，工作线程删除数组prime[]中base的倍数。（注意，我们在main()开头就消除了2的所有倍数，因而仅需要考虑base的奇数值。）

工作线程知道要使用的base的值后，就可以从共享的数组prime[]中安全地删去base的倍数，因为没有其他工作线程会使用这个base值。然而，必须将*防护语句*放在base基于的共享变量nextbase的更新操作周围（第26行）。记住，任何工作线程都会被另一个工作线程在不可预测的时间抢占，而抢占的工作线程会在worker()的代码中不可预测的位置。特别是，可能碰巧在如下语句的中间中断当前工作线程：

```
base = nextbase += 2;
```

而且下一个时间片被赋予了也执行这个语句的另一个线程。在这种情况下，有两个工作线程试图立即修改共享变量nextbase，它可能导致隐伏且难以重现的程序错误。

用防护语句将操作共享变量的代码（称为关键部分）括起来，可以防止发生这种情况。对pthread_mutex_lock()和pthread_mutex_unlock()的调用可以确保基本只有一个线程执行括起来的程序片段。对这两个函数的调用告诉操作系统，只有当前没有其他线程在执行关键部分时，才允许一个线程进入这个关键部分，并且让操作系统不要抢占该线程，直到它完成整个部分。（线程系统内部使用锁变量nextbaselock来确保这种“互斥现象”。）

但是，往往太容易发生这样的情况：没有成功地识别和/正确地保护线程代码中的关键部分。让我们看看如何使用GDB调试Pthreads程序中的这种错误。假设我们忘记包括下面这个解锁语句：

```
pthread_mutex_unlock(&nextbaselock);
```

这样，一旦关键部分被一个工作线程先进入，而另一个工作线程在永久等待锁被释放，肯定会导致程序挂起。但是让我们假设你不知道这一点。如何使用GDB找出故障发生的地方呢？

编译程序，确保加上了-lpthread-lm标记，以便链接Pthreads和数学函数库（调用sqrt()需要后者）。然后运行GDB中的代码，使n=100且nthreads=2。

```
(gdb) r 100 2
Starting program: /debug/primes 100 2
[New Thread 16384 (LWP 28653)]
[New Thread 32769 (LWP 28676)]
[New Thread 16386 (LWP 28677)]
[New Thread 32771 (LWP 28678)]
```

每次创建新线程时，GDB都会宣布一下，如上所示。我们很快会带读者观察哪个线程在做什么。

程序挂起了，通过按下Ctrl+C组合键中断它。GDB会话如下所示。

```
(gdb) r 100 2
Starting program: /debug/primes 100 2
[New Thread 16384 (LWP 28653)]
[New Thread 32769 (LWP 28676)]
[New Thread 16386 (LWP 28677)]
[New Thread 32771 (LWP 28678)]

Program received signal SIGINT, Interrupt.
[Switching to Thread 32771 (LWP 28678)]
0x4005ba35 in __pthread_sigsuspend () from /lib/i686/libpthread.so.0
```

在这种时候，关键要知道每个线程在做什么，可以通过GDB的info threads命令来确定：

```
(gdb) info threads
* 4 Thread 32771 (LWP 28678)  0x4005ba35 in __pthread_sigsuspend ()
   from /lib/i686/libpthread.so.0
  3 Thread 16386 (LWP 28677)  0x4005ba35 in __pthread_sigsuspend ()
   from /lib/i686/libpthread.so.0
  2 Thread 32769 (LWP 28676)  0x420db1a7 in poll () from
```

```
/lib/i686/libc.so.6
  1 Thread 16384 (LWP 28653)  0x4005ba35 in __pthread_sigsuspend ()
   from /lib/i686/libpthread.so.0
```

星号表示当前在线程4中。让我们看看那个线程是做什么的。

```
(gdb) bt
#0 0x4005ba35 in __pthread_sigsuspend () from /lib/i686/libpthread.so.0
#1 0x4005adb8 in __pthread_wait_for_restart_signal ()
   from /lib/i686/libpthread.so.0
#2 0x4005d190 in __pthread_alt_lock () from /lib/i686/libpthread.so.0
#3 0x40059d77 in pthread_mutex_lock () from /lib/i686/libpthread.so.0
#4 0x0804855f in worker (tn=1) at Primes.c:49
#5 0x40059881 in pthread_start_thread () from /lib/i686/libpthread.so.0
#6 0x40059985 in pthread_start_thread_event () from
/lib/i686/libpthread.so.0
```

（这在Pthreads的LinuxThreads实现下是可以做到的，但是在其他平台上不一定起作用。）

从该代码中看到在帧3和帧4中，这个线程在源代码的第49行上，并且在试图获得锁和进入关键部分。

```
pthread_mutex_lock(&nextbaselock);
```

还要注意，从帧0向上的线程当前显然被挂起了，等待另一个线程释放锁。除非发生释放锁的情况并且线程管理程序安排它获得锁，否则这个线程不会得到任何时间片。

其他线程在做什么呢？可以通过切换到其他任何线程然后执行bt命令来检查该线程的栈。

```
(gdb) thread 3
[Switching to thread 3 (Thread 16386 (LWP 28677))]#0  0x4005ba35 in
__pthread_sigsuspend () from /lib/i686/libpthread.so.0
(gdb) bt
#0  0x4005ba35 in __pthread_sigsuspend () from /lib/i686/libpthread.so.0
#1  0x4005adb8 in __pthread_wait_for_restart_signal ()
   from /lib/i686/libpthread.so.0
#2  0x4005d190 in __pthread_alt_lock () from /lib/i686/libpthread.so.0
#3  0x40059d77 in pthread_mutex_lock () from /lib/i686/libpthread.so.0
#4  0x0804855f in worker (tn=0) at Primes.c:49
#5  0x40059881 in pthread_start_thread () from /lib/i686/libpthread.so.0
#6  0x40059985 in pthread_start_thread_event () from
/lib/i686/libpthread.so.0
```

我们曾创建过两个工作线程。上面看到的线程4是其中之一（bt输出中的帧4），现在从这里输出的帧4中看到线程3是另一个工作线程。还可以看到线程3也在试图获得锁（帧3）。

虽然不应有任何其他工作线程，但是调试的一个基本原则是不可以无条件相信任何事情，一切都必须检查。我们现在通过检查其余线程的状态来做这件事。你将发现，其他两个线程现在是非工作线程，如下所示。

```
(gdb) thread 2
[Switching to thread 2 (Thread 32769 (LWP 28676))]#0  0x420db1a7 in poll
()
   from /lib/i686/libc.so.6
(gdb) bt
#0  0x420db1a7 in poll () from /lib/i686/libc.so.6
#1  0x400589de in __pthread_manager () from /lib/i686/libpthread.so.0
#2  0x4005962b in __pthread_manager_event () from
/lib/i686/libpthread.so.0
```

因此线程2是线程管理程序。它是Pthreads包的内在线程。它当然不是工作线程，这一点部分地确认了我们对只有两个工作线程的预期。检查线程1：

```
(gdb) thread 1
[Switching to thread 1 (Thread 16384 (LWP 28653))]#0  0x4005ba35 in
__pthread_sigsuspend () from /lib/i686/libpthread.so.0
(gdb) bt
#0  0x4005ba35 in __pthread_sigsuspend () from /lib/i686/libpthread.so.0
#1  0x4005adb8 in __pthread_wait_for_restart_signal ()
   from /lib/i686/libpthread.so.0
#2  0x40058551 in pthread_join () from /lib/i686/libpthread.so.0
#3  0x080486aa in main (argc=3, argv=0xbfffe7b4) at Primes.c:83
#4  0x420158f7 in __libc_start_main () from /lib/i686/libc.so.6
```

发现它是执行`main()`的，因此可以确认只有两个工作线程。

然而，两个工作线程都停止了，每个线程都在等待释放锁。难怪程序挂起了！这样足以查明程序错误的位置和性质，我们很快意识到是忘记了调用解锁函数。

5.2.3　变体

如果在一开始没有意识到防护共享变量`nextbase`的必要性，会发生什么事情？如果在前面的示例中遗漏了解锁和加锁操作，会发生什么情况？

乍一看这个问题，可能会以为对程序的正确运行没有影响（即得到精确的素数数量），只不过可能会由于重复的工作而使运行变慢（即使用`base`的相同值多于一次）。似乎有些线程会重复其他线程的工作，如两个工作线程恰好抢夺`nextbase`的同一个值来初始化它们的`base`本地副本。然后，有些合数可能最后会被删去两次，但是结果（即素数的数量）仍然是正确的。

但是让我们仔细看一下。语句

```
base = nextbase += 2;
```

至少编译成两条机器语言指令。例如，在运行Linux系统的Pentium机器上使用GCC编译器，上面的C语句翻译成如下汇编语言指令（通过用`-S`选项运行GCC，然后查看得到的*.s*文件来获得汇编语言指令）：

```
addl $2, nextbase
movl nextbase, %eax
movl %eax, -8(%ebp)
```

该代码将nextbase递增2，然后将nextbase的值复制到寄存器EAX中，最后，将EAX的值复制到工作线程的栈中存储本地变量base的位置。

假设只有两个工作线程，并且假设nextbase的值为9，那么当前正在运行的worker()调用的时间片在执行如下机器指令后会立即结束：

```
addl $2, nextbase
```

该指令将共享全局变量nextbase设置为11。假设下一个时间片给予了worker()的另一个调用，它恰好在执行那些相同的指令。那么第二个工作线程现在将nextbase递增到13，使用这个值来设置其局部变量base，并开始消除所有13的倍数。最后，worker()的第一个调用会获得另一个时间片，然后从停止的地方继续执行机器指令：

```
movl nextbase, %eax
movl %eax, -8(%ebp)
```

当然，nextbase的值现在是13。因此，第一个工作线程将它的局部变量base的值设置为13，并继续消除这个值的倍数，而不是设置为它在其上一个时间片期间设置的值11。两个工作线程都不会用11的倍数做任何事。最终不仅毫无必要地重复了工作，而且忽略了必需的工作！

如何使用GDB发现这样的错误呢？表面“症状”可能是报告的素数数量太大。因此，你可能猜测base的值不知何故有时被跳过了。为了检查这一假设，可以在下面这行代码后面放置一个断点。

5

```
base = nextbase += 2;
```

通过重复执行GDB的continue (c)命令，并显示base的值，

```
(gdb) disp base
```

最终可能确认base的值真的被跳过了。

这里的关键词是“可能”。我们前面讨论过，多线程程序的运行有一定的随机性。在这个例子里，有可能出现这种情况：有时运行程序后会出现程序错误（即报告的素数太多），但是有时运行程序后又可能得到了正确的答案！

然而，对这种问题没有很好的解决方案。调试多线程代码常常需要特别有耐心和创意。

5.2.4 GDB线程命令汇总

下面是与线程相关的GDB命令用法汇总：

- info threads（给出关于当前所有线程的信息）；

- `thread 3`（改成线程3）；
- `bread 88 thread 3`（当线程3到达源代码行88时停止执行）；
- `break 88 thread 3 if x==y`（当线程3到达源代码行88，并且变量*x*和*y*相等时停止执行）。

5.2.5　DDD中的线程命令

如图5-1所示，在DDD中选择Status→Threads会弹出一个窗口，按GDB的`info threads`方式显示所有线程。可以单击一个线程将调试器的焦点切换到这个线程上。

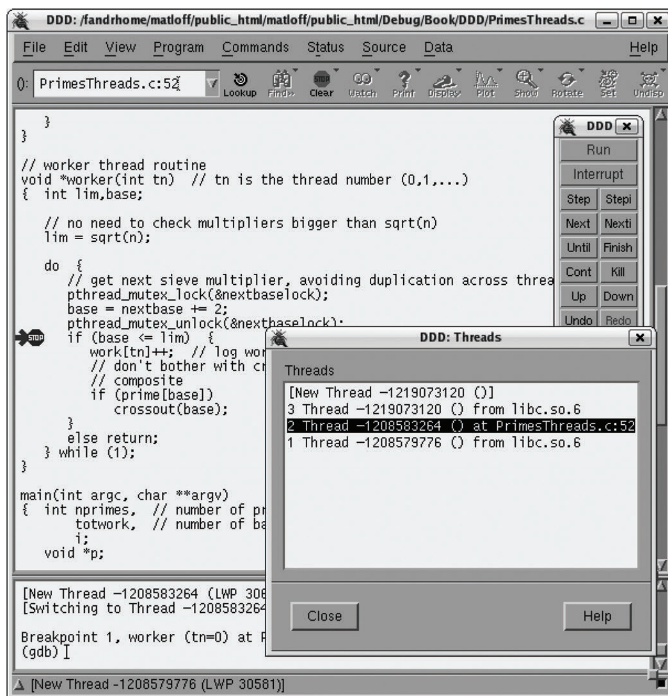


图5-1　线程窗口

最好保持这个弹出窗口打开，而不是使用一次就关闭。这样，就不必总是在每次要查看当前有哪些线程在运行或者切换到不同的线程时都重新打开这个窗口。

DDD中似乎没有办法使上面介绍的GDB命令`break 88 thread 3`对特定线程设置断点，而是通过DDD控制台来执行这样的GDB命令。

5.2.6　Eclipse中的线程命令

首先要注意，Eclipse创建的默认make文件不会包括GCC的`-lpthread`命令行参数（也不包括所需的其他任何特殊库的参数）。如果你愿意，可以直接更改make文件，让Eclipse做这件事更容易。在C/C++透视图中右击项目名，单击Properties；选择C/C++ Build旁边的三角形图标，使其变为下三角；选择Settings→Tool Settings；选择GCC C Linker旁边的三角形图标，使其成

为下三角，再选择Libraries→Add（后者是绿色的加号图标）；并在`-l`后面填上库标志（比如，填写`m`组成`-lm`）。然后构建项目。

第1章介绍过，Eclipse不断地显示线程列表，而不像DDD要请求才显示。而且，不需要像在DDD中那样请求回溯操作，调用栈显示在图5-2所示的线程列表中。正如上面所介绍的，我们先运行一会儿程序，然后单击*Resume*右边的Suspend图标中断它。线程列表在Debug视图中，它一般在屏幕的左上方，但是这里以展开的形式出现，因为我们单击了Debug选项卡中的Maximize。（可以单击Restore回到标准布局。）

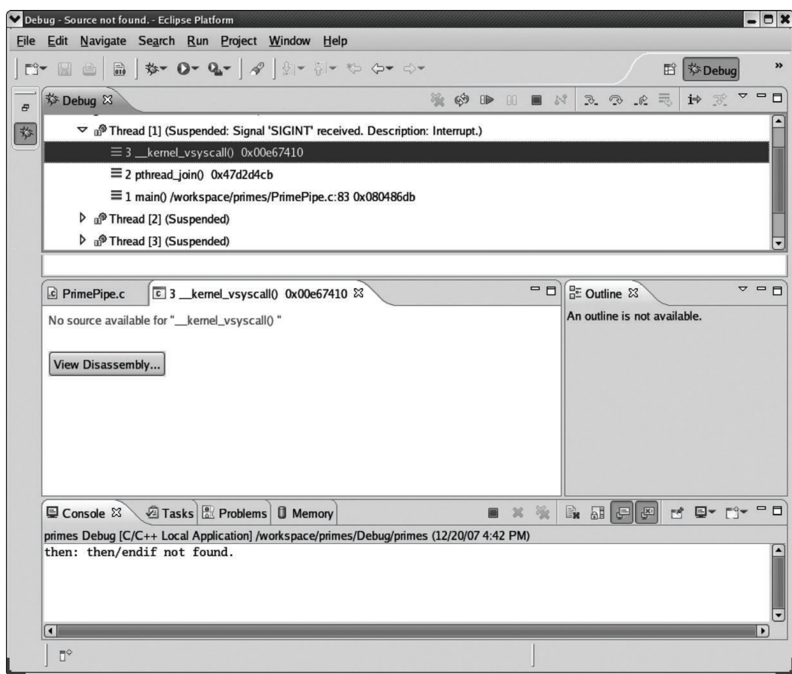


图5-2 Eclipse中的线程显示

我们发现，中断发生时线程3正在运行，它收到一个SIGINT信号，这是中断（CTRL+C）信号。我们还看到相关的系统调用通过`pthread_join()`调用，而后者由`main()`调用。从该程序以前的情况来看，我们知道它实际上是主线程。

要查看另一个线程的信息，只要把该线程旁边的图标点成下三角形即可。要改成另一个线程，单击列表中另一个线程对应的项。

我们可能要设置仅适用于一个特定线程的断点。要做到这一点，必须先等到创建了该线程。然后，当前面设置的断点暂停程序执行，或者出现上面介绍的中断时，我们用创建断点条件的方式右击断点符号，但是这次选择的是Filtering。这时会出现一个类似图5-3所示的弹出窗口。我们看到，这个断点当前应用于所有3个线程。例如，如果希望这个断点仅应用于线程2，只要点击其他两个线程项旁边的复选框取消即可。

5

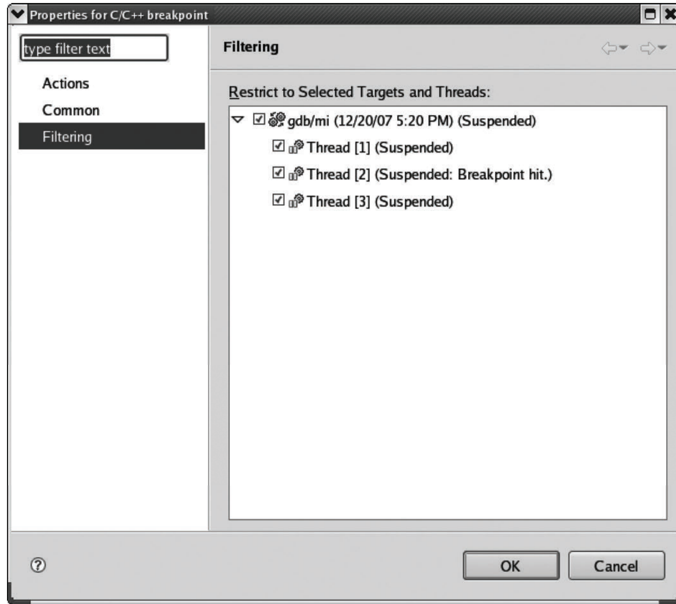


图5-3　在Eclipse中设置某个线程特有的断点

5.3　调试并行应用程序

并行编程架构主要有两种：共享内存和消息传递。

术语共享内存的确切含义是：多个CPU都具有对某些共同的物理内存的访问权限。在一个CPU上运行的代码与在其他CPU上运行的代码通信，方法是通过在这个共享内存上读写。这与多线程应用程序中的线程通过共享地址空间与另一个线程通信基本一样。（事实上，多线程编程变成了为共享内存系统编写应用程序代码的标准方式。）

相反，在消息传递环境下，在各个CPU上运行的代码只能访问该CPU的本地内存，它通过通信媒介上发送称为“消息”的字节串来与其他CPU上的代码通信。通常这是某种网络，运用某种通用协议（比如TCP/IP）或者适用于消息传递应用程序的专门软件基础结构。

5.3.1　消息传递系统

本节首先以流行的MPI（Message Passing Interface）包为例，讨论消息传递。我们这里使用的是MPICH实现，但是同样的原理也适用于LAM和其他MPI实现。

让我们再来看一个求素数的程序。

```
1   #include <mpi.h>
2
3   // MPI sample program; not intended to be efficient; finds and reports
4   // the number of primes less than or equal to n
5
```

```
6   // Uses a pipeline approach:  node 0 looks at all the odd numbers (i.e.,
7   // we assume multiples of 2 are already filtered out) and filters out
8   // those that are multiples of 3, passing the rest to node 1; node 1
9   // filters out the multiples of 5, passing the rest to node 2; node 2
10  // filters out the rest of the composites and then reports the number
11  // of primes
12
13  // the command-line arguments are n and debugwait
14
15  #define PIPE_MSG 0  // type of message containing a number to
16                      // be checked
17  #define END_MSG 1  // type of message indicating no more data will
18                     // be coming
19
20  int nnodes,  // number of nodes in computation
21      n,  // find all primes from 2 to n
22      me;  // my node number
23
24  init(int argc,char **argv)
25  {  int debugwait;  // if 1, then loop around until the
26                     // debugger has been attached
27
28     MPI_Init(&argc,&argv);
29     n = atoi(argv[1]);
30     debugwait = atoi(argv[2]);
31
32     MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
33     MPI_Comm_rank(MPI_COMM_WORLD,&me);
34
35     while (debugwait) ;
36  }
37
38  void node0()
39  {  int i,dummy,
40         tocheck; // current number to check for passing on to next node
41     for (i = 1; i <= n/2; i++)  {
42        tocheck = 2 * i + 1;
43        if (tocheck > n) break;
44        if (tocheck % 3 > 0)
45           MPI_Send(&tocheck,1,MPI_INT,1,PIPE_MSG,MPI_COMM_WORLD);
46     }
47     MPI_Send(&dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
48  }
49
50  void node1()
```

```
51  {  int tocheck,  // current number to check from node 0
52         dummy;
53     MPI_Status status;  // see below
54
55     while (1)  {
56        MPI_Recv(&tocheck,1,MPI_INT,0,MPI_ANY_TAG,
57           MPI_COMM_WORLD,&status);
58        if (status.MPI_TAG == END_MSG) break;
59        if (tocheck % 5 > 0)
60           MPI_Send(&tocheck,1,MPI_INT,2,PIPE_MSG,MPI_COMM_WORLD);
61     }
62     // now send our end-of-data signal, which is conveyed in the
63     // message type, not the message itself
64     MPI_Send(&dummy,1,MPI_INT,2,END_MSG,MPI_COMM_WORLD);
65  }
66
67  void node2()
68  {  int tocheck,  // current number to check from node 1
69         primecount,i,iscomposite;
70     MPI_Status status;
71
72     primecount = 3;  // must account for the primes 2, 3 and 5, which
73                      // won't be detected below
74     while (1)  {
75        MPI_Recv(&tocheck,1,MPI_INT,1,MPI_ANY_TAG,
76           MPI_COMM_WORLD,&status);
77        if (status.MPI_TAG == END_MSG) break;
78        iscomposite = 0;
79        for (i = 7; i*i <= tocheck; i += 2)
80           if (tocheck % i == 0)  {
81              iscomposite = 1;
82              break;
83           }
84        if (!iscomposite) primecount++;
85     }
86     printf("number of primes = %d\n",primecount);
87  }
88
89  main(int argc,char **argv)
90  {  init(argc,argv);
91     switch (me)  {
92        case 0:  node0();
93            break;
94        case 1:  node1();
95            break;
```

```
96          case 2:  node2();
97       };
98       MPI_Finalize();
99    }
```

正如本程序开头的注释所解释的，这里，埃拉托色尼筛法（找素数算法）在并行系统的3个节点上运行，以多通道的方式工作。第一个节点从奇数开始，去掉所有3的倍数，让其余值通过筛选；第二个节点获得第一个节点的输出，并去掉所有5的倍数；第三个节点获得第二个节点的输出，去掉余下的所有非素数，并报告剩下的素数数量。

这里，通过让每个节点向下一个节点一次传递一个数字来获得流水线操作。（在各条MPI消息中传递成组的数字因而减少通信开销，效率会高得多。）当向下一个节点发送数字时，节点发送PIPE_MSG类型的消息。当节点没有更多数字要发送时，会发送一条END_MSG类型的消息予以说明。

作为这里的一个调试示例，假设我们忘记了在第一个节点中包括后面的通知，即忘记了第46行代码的node0()。

```
MPI_Send(&dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
```

该程序会在“下游”节点处挂起。让我们看看如何跟踪这个程序错误。（记住，下面的GDB会话中有些行号会与上面的代码清单中相差1。）

在系统的一个节点上调用名为mpirun的脚本，从而运行MPICH应用程序。该脚本通过SSH在各个节点启动应用程序。这里我们在一个有3台机器的网络上做这件事，3台机器分别称为节点0、节点1和节点2，n等于100。程序错误导致程序在后两个节点处挂起。程序也在第一个节点处挂起，因为只有MPI程序的所有实例都执行了MPI_FINALIZE()函数之后，才有实例会退出。

5

虽然我们打算使用GDB，但是因为在这3个节点上都使用了mpirun来调用应用程序，而不是直接在节点上运行它们，因此我们不能直接运行GDB。然而，GDB允许使用进程号动态地将调试器附加到已经运行的进程上。因此，让我们在节点1上运行ps来确定正在执行应用程序的进程。

```
$ ps ax
...
 2755 ?        S      0:00 tcsh -c /home/matloff/primepipe node 1 3
 2776 ?        S      0:00 /home/matloff/primepipe node1 32812    4
 2777 ?        S      0:00 /home/matloff/primepipe node1 32812    4
```

MPI程序在运行进程2776，因此将GDB附加到节点1的程序上。

```
$ gdb primepipe 2776
...
0xffffe002 in ?? ()
```

该代码提供的信息不是太丰富。因此，让我们看看现在位于何处。

```
(gdb) bt
#0  0xffffe002 in ?? ()
#1  0x08074a76 in recv_message ()
#2  0x080748ad in p4_recv ()
#3  0x0807ab46 in MPID_CH_Check_incoming ()
#4  0x08076ae9 in MPID_RecvComplete ()
#5  0x0806765f in MPID_RecvDatatype ()
#6  0x0804a29f in PMPI_Recv ()
#7  0x08049ce8 in node1 () at PrimePipe.c:56
#8  0x08049e19 in main (argc=8, argv=0xbffffb24) at PrimePipe.c:94
#9  0x420156a4 in __libc_start_main () from /lib/tls/libc.so.6
```

从帧7处看到程序在第56行处挂起，等待从节点0接收输出。

接下来，知道在节点1（`node1()`）上运行的函数完成了多少工作会是有用的。是刚刚开始，还是已经完成？可以通过确定变量`tocheck`的上一个已处理的值来衡量进度。

```
(gdb) frame 7
#7  0x08049ce8 in node1 () at PrimePipe.c:56
56          MPI_Recv(&tocheck,1,MPI_INT,0,MPI_ANY_TAG,
(gdb) p tocheck
$1 = 97
```

说明　需要先使用GDB的`frame`命令移动到`node1()`的栈帧。

这段代码表明节点1已执行结束，因为97应当是节点0传递给这个节点进行素数检查的最后一个数字。因此，目前我们料到从节点0处会产生一条END_MSG类型的消息。程序挂起这一事实暗示了节点0可能没有发送这样一条消息，而这又导致我们检查它是否具有这样的消息。按这种方式，有望很快找到程序错误，原来是因为不小心漏掉了第46行。

顺便说一下，当像上面那样使用如下命令：

```
$ gdb primepipe 2776
```

调用GDB时，GDB的命令行首先检查名为2776的核心文件。万一存在这样的文件，GDB会加载这个文件，而不是附加到目标进程后面。另外，GDB也有`attach`命令。

在本例中，程序错误导致程序挂起。调试像本例这样的并行程序的方法与出现错误输出征兆时的方法略有不同。例如，假设在第71行错误地将`primecount`初始化成了2而不是3。如果试图采用相同的调试过程，在每个节点上运行的程序就会很快完成执行并退出，而来不及附加GDB。（确实，可以使用非常大的`n`值，但是通常一开始先用简单的情况调试会更好。）我们需要某种可用来让程序等待让我们附加GDB的机制，而这正是第34行`init()`函数的作用所在。

正如在源代码中看到的，`debugwait`的值取自用户提供的命令行，1表示等待，2表示不等待。如果将`debugwait`的值指定为1，那么每次调用程序时到达第34行，它都会在那里。这样就为我们

提供了附加GDB的时间。然后可以跳出无限循环，并继续调试。下面是在节点0处所做的。

```
node1:~$ gdb primepipe 3124
...
0x08049c53 in init (argc=3, argv=0xbfffe2f4) at PrimePipe.c:34
34          while (debugwait) ;
(gdb) set debugwait = 0
(gdb) c
Continuing.
```

我们一般都怕无限循环，但是这里为了便于调试，却故意建立了一个无限循环在节点1和节点2处做相同的事情，在后一个节点上，还在继续调试之前利用了在第77行设置断点的机会。

```
[matloff@node3 ~]$ gdb primepipe 2944
34          while (debugwait) ;
(gdb) b 77
Breakpoint 1 at 0x8049d7d: file PrimePipe.c, line 77.
(gdb) set debugwait = 0
(gdb) c
Continuing.

Breakpoint 1, node2 () at PrimePipe.c:77
77             if (status.MPI_TAG == END_MSG) break;
(gdb) p tocheck
$1 = 7
(gdb) n
78             iscomposite = 0;
(gdb) n
79             for (i = 7; i*i <= tocheck; i += 2)
(gdb) n
84             if (!iscomposite) primecount++;
(gdb) n
75             MPI_Recv(&tocheck,1,MPI_INT,1,MPI_ANY_TAG,
(gdb) p primecount
$2 = 3
```

这时，注意到`primecount`应为4，而不是3（7以下的素数是2、3、5和7），因此找到了程序错误的位置。

5.3.2 共享内存系统

本节介绍共享内存类型的并行编程情形。下面我们将真正的共享内存机器与软件分布式共享内存设置的情况分开介绍。

1. 真正的共享内存

正如前面所提到的，在真正的共享内存环境中，通常使用线程来开发应用程序。5.2节介绍

的用GDB/DDD调试的内容就派上了用场。

OpenMP是这样的机器上流行的编程环境。OpenMP向程序员提供了使用线程的高级并行编程结构。如有必要，程序员仍然具有线程级的访问权限，但是在大多数情况下，程序员不需要了解OpenMP指令的多线程实现细节。

5.4节将提供一个关于调试OpenMP应用程序的扩展示例。

2. 软件分布式共享内存系统

虽然双核CPU机器现在普通消费者也买得起了，但是具有多个处理器的大型共享内存系统仍然要花几十万美元才能买到。一种流行的、不算太贵的替代品是*工作站网络*（network of workstations，NOW）。NOW架构使用了可以造成共享内存错觉的底层库。应用程序员基本上不需要了解这个库，它主要从事维护不同节点之间共享变量的副本统一性这样的网络事务。

这种方法称为*软件分布式共享内存*（SDSM）。用得最广泛的SDSM库是莱斯大学（Rice University）开发并维护的Treadmarks。另一个优秀的软件包是JIAJIA，可以从何田的站点（*http://www-users.cs.umn.edu/~tianhe/paper/dist.htm*）下载。

SDSM应用程序展示的某种行为会打击粗心的程序员。它们高度依赖于特定的系统，因此这里无法给出通用的对待方法，但是我们将简要介绍其中的一些共同问题。

很多SDSM是基于页的，这意味着它们依赖于节点上的底层虚拟内存硬件。虽然动作比较复杂，但是我们可以给出一个快速概览。假设要在NOW节点之间共享变量x。程序员是这样表达这个意图的：对SDSM库进行某个调用，这个库再发出一个UNIX系统调用，请求操作系统对于涉及包含x的页错误，使用SDSM库中的函数替换它自己的段错误处理程序。SDSM的工作方式是：只有具有x的有效副本的NOW节点才具有标记为驻留的对应内存页。当在其他节点上访问x时，就会产生页错误，底层SDSM软件从拥有x的节点处取得正确的值。

同样地，没有必要知道SDSM系统的确切运作；相反，要知道有一个底层的基于虚拟内存的机制，在维护各NOW节点的共享数据的本地副本一致性。如果不了解这一点，那么当试图调试SDSM应用程序代码时可能会丈二和尚摸不着头脑。调试器可能会由于不存在的段错误神秘地停止，因为SDSM基础设施专门产生段错误，当SDSM应用程序在调试工具下运行时，调试工具能感知它们。意识到这一点后就根本没有问题，在GDB中，当发生这种奇怪的暂停情况时，只要执行`continue`命令就可以恢复执行。

每当发生段错误时，你可能会试图使用如下GDB命令来命令GDB不要停止，或让它发出警告消息：

```
handle SIGSEGV nostop noprint
```

不过，要慎用这种方法，因为这样可能导致漏掉应用程序中的程序错误引起的一些真正的段错误。

然而，针对基于页的SDSM上运行调试程序时还会出现另一个困难。如果网络上的一个节点修改了共享变量的值，那么需要该变量的其他任何节点都必须通过网络事务获得更新后的值。再次说明，这件事的具体发生方法取决于SDSM系统，但是这意味着，单步调试在一个执行点执行

代码时，可能发现GDB神秘地挂起，因为节点正在等待另一个节点最近修改过的变量的本地副本。如果你碰巧还在运行一个独立的GDB会话，以单步调试另一个节点上的代码，那么直到第二个节点上的调试会话有了足够的进度后，第一个节点上才会发生更新。换言之，如果在SDSM应用程序调试期间程序员不够警觉和小心，那么他可能由于调试进程本身使自己陷入死锁。

SDSM情况在某种意义上类似于消息传递的情况：需要类似上面的MPI示例中的`debugwait`变量。该变量允许程序在所有节点上暂停，给了程序员在每个节点上附加GDB并从头开始单步调试程序的机会。

5.4 扩展示例

本节介绍一个调试用OpenMP开发的共享内存应用程序的示例。下面先介绍OpenMP的必备知识，只要对线程有基本了解即可。

5.4.1 OpenMP概述

OpenMP本质上是线程管理操作的高级并行编程接口。线程数量通过环境变量`OMP_NUM_THREADS`设置。例如，在C Shell下，在shell提示符下键入：

```
% setenv OMP_NUM_THREADS 4
```

可以安排4个线程。由C语言组成的应用程序代码中散布着OpenMP指令。每个指令适用于该指令后面的块，用左右花括号定界。最基本的指令为：

```
#pragma omp parallel
```

这个指令建立了`OMP_NUM_THREADS`线程，每个线程并发执行pragma后面的代码块。这个块中通常会嵌入其他指令。

另一个非常常见的OpenMP指令是：

```
#pragma omp barrier
```

该指令指定所有线程的“会合点”。任一线程达到这一点时就会阻塞，直到其他线程全部到达那里。

如果希望只有一个线程执行某个块，而其他线程跳过这个块，可以通过编写如下代码来实现。

```
#pragma omp single
```

这样的块后面立即有一个隐含的障碍。

虽然有很多其他OpenMP指令，但是本例中使用的另一个指令是：

```
#pragma omp critical
```

顾名思义，该指令创建了一个关键部分，其中在任何给定时间只允许一个线程。

5.4.2 OpenMP示例程序

我们实现著名的Dijkstra算法来确定加权图中一对顶点之间的最小距离。给出相邻顶点之间的距离（如果两个顶点不相邻，那么它们之间的距离被设置为无穷大）。目标是求顶点0和其他所有顶点之间的最小距离。

下面是源文件*dijkstra.c*。该程序产生指定数目的顶点之间的随机边长，然后求顶点0到其他各顶点之间的最小距离。

```
1  // dijkstra.c
2
3  // OpenMP example program:  Dijkstra shortest-path finder in a
4  // bidirectional graph; finds the shortest path from vertex 0 to all
5  // others
6
7  // usage:  dijkstra nv print
8
9  // where nv is the size of the graph, and print is 1 if graph and min
10 // distances are to be printed out, 0 otherwise
11
12 #include <omp.h>  // required
13 #include <values.h>
14
15 // including stdlib.h and stdio.h seems to cause a conflict with the
16 // Omni compiler, so declare directly
17 extern void *malloc();
18 extern int printf(char *,...);
19
20 // global variables, shared by all threads
21 int nv,  // number of vertices
22     *notdone, // vertices not checked yet
23     nth,  // number of threads
24     chunk,  // number of vertices handled by each thread
25     md,  // current min over all threads
26     mv;  // vertex which achieves that min
27
28 int *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
29            // ohd[i*nv+j]
30     *mind;  // min distances found so far
31
32 void init(int ac, char **av)
33 {  int i,j,tmp;
34    nv = atoi(av[1]);
35    ohd = malloc(nv*nv*sizeof(int));
36    mind = malloc(nv*sizeof(int));
37    notdone = malloc(nv*sizeof(int));
```

```
38      // random graph
39      for (i = 0; i < nv; i++)
40         for (j = i; j < nv; j++)  {
41            if (j == i) ohd[i*nv+i] = 0;
42            else  {
43               ohd[nv*i+j] = rand() % 20;
44               ohd[nv*j+i] = ohd[nv*i+j];
45            }
46         }
47      for (i = 1; i < nv; i++)  {
48         notdone[i] = 1;
49         mind[i] = ohd[i];
50      }
51   }
52
53   // finds closest to 0 among notdone, among s through e; returns min
54   // distance in *d, closest vertex in *v
55   void findmymin(int s, int e, int *d, int *v)
56   {  int i;
57      *d = MAXINT;
58      for (i = s; i <= e; i++)
59         if (notdone[i] && mind[i] < *d)  {
60            *d = mind[i];
61            *v = i;
62         }
63   }
64
65   // for each i in {s,...,e}, ask whether a shorter path to i exists, through
66   // mv
67   void updatemind(int s, int e)
68   {  int i;
69      for (i = s; i <= e; i++)
70         if (notdone[i])
71            if (mind[mv] + ohd[mv*nv+i] < mind[i])
72               mind[i] = mind[mv] + ohd[mv*nv+i];
73   }
74
75   void dowork()
76   {
77      #pragma omp parallel
78      {  int startv,endv,  // start, end vertices for this thread
79              step,  // whole procedure goes nv steps
80              mymv,  // vertex which attains that value
81              me = omp_get_thread_num(),
82              mymd;  // min value found by this thread
```

```
83          #pragma omp single
84          {  nth = omp_get_num_threads();  chunk = nv/nth;
85             printf("there are %d threads\n",nth);  }
86          startv = me * chunk;
87          endv = startv + chunk - 1;
88          // the algorithm goes through nv iterations
89          for (step = 0; step < nv; step++)  {
90             // find closest vertex to 0 among notdone; each thread finds
91             // closest in its group, then we find overall closest
92             #pragma omp single
93             {  md = MAXINT;
94                mv = 0;
95             }
96             findmymin(startv,endv,&mymd,&mymv);
97             // update overall min if mine is smaller
98             #pragma omp critical
99             {  if (mymd < md)
100                  {  md = mymd;  }
101            }
102            #pragma omp barrier
103            // mark new vertex as done
104            #pragma omp single
105            {  notdone[mv] = 0;  }
106            // now update my section of ohd
107            updatemind(startv,endv);
108         }
109      }
110   }
111
112   int main(int argc, char **argv)
113   {  int i,j,print;
114      init(argc,argv);
115      // start parallel
116      dowork();
117      // back to single thread
118      print = atoi(argv[2]);
119      if (print)  {
120         printf("graph weights:\n");
121         for (i = 0; i < nv; i++)  {
122            for (j = 0; j < nv; j++)
123               printf("%u  ",ohd[nv*i+j]);
124            printf("\n");
125         }
126         printf("minimum distances:\n");
127         for (i = 1; i < nv; i++)
```

```
128            printf("%u\n",mind[i]);
129        }
130    }
```

让我们回顾一下该算法的工作原理。从除顶点0外的所有顶点开始，在本例中是“未完成”集合中的顶点1～5。在该算法的每次迭代中，进行如下操作。

(1) 沿着到目前为止的已知路径求得离顶点0最近的“未完成”顶点v。这种检查被所有线程共享，每个线程检查相等数量的顶点。完成这一工作的函数是`findmymin()`。

(2) 然后将v移到“已完成”集合。

(3) 对于“未完成”集合中余下的所有顶点`i`，检查沿着到目前为止的已知路径先从0到`v`，然后一下子从v跳到`i`，是否比从0到`i`的当前已知最短距离更短。如果是这样，相应地更新该距离。执行这些动作的函数是`updatemind()`。

该迭代继续，直到“未完成”集合为空。

由于OpenMP指令需要预处理，因此总是有失去原来的行号和变量及函数名的潜在问题。为了了解如何解决这个问题，我们将讨论两个不同编译器的情况。我们首先讨论Omni编译器（*http://www.hpcc.jp/Omni/*），然后讨论GCC（需要4.2或更新的版本）。

在Omni下对代码展开编译：

```
$ omcc -g -o dij dijkstra.c
```

当编译该程序并用4个线程运行它以后，我们发现它没有正确地工作。

```
$ dij 6 1
there are 4 threads
graph weights:
0  3  6  17  15  13
3  0  15  6  12  9
6  15  0  1  2  7
17  6  1  0  10  19
15  12  2  10  0  3
13  9  7  19  3  0
minimum distances:
3
6
17
15
13
```

手工分析这个图可以发现，正确的最小距离应为3、6、7、8和11。

然后，我们在GDB中运行程序。理解OpenMP通过指令工作这一事实的后果十分重要。虽然这里讨论的两个编译器基本都保留了行号、函数名等，但是两者之间仍然有一些差异。当试图在可执行文件`dij`中设置断点时，看看在我们的GDB会话开端发生了什么。

```
(gdb) tb main
Breakpoint 1 at 0x80492af
(gdb) r 6 1
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1208490304 (LWP 11580)]
[Switching to Thread -1208490304 (LWP 11580)]
0x080492af in main ()
(gdb) l
1       /tmp/omni_C_11486.c: No such file or directory.
        in /tmp/omni_C_11486.c
```

我们发现断点不在源文件中，而是在Omni的OpenMP基础设施代码中。换言之，这里的main()是Omni的main()，而不是程序员自己的main()。Omni编译器将我们的main()名称变成了_ompc_main()。

为了在main()上设置断点，键入：

```
(gdb) tb _ompc_main
Breakpoint 2 at 0x80491b3: file dijkstra.c, line 114.
```

然后通过continuing来检查它：

```
(gdb) c
Continuing.
[New Thread -1208493152 (LWP 11614)]
[New Thread -1218983008 (LWP 11615)]
[New Thread -1229472864 (LWP 11616)]
_ompc_main (argc=3, argv=0xbfab6314) at dijkstra.c:114
114         init(argc,argv);
```

这段代码中有熟悉的init()代码。当然，可以执行命令：

```
(gdb) b dijkstra.c:114
```

注意3个新线程的创建，总共是4个线程。

然而，我们选择了设置断点，在这里我们必须比一般情况多做一点工作，因此在程序的两次运行期间停留在一个GDB会话中特别重要，甚至当我们修改了源代码并重新编译时也不要退出GDB会话，以便保留断点、条件等。如果在这期间关闭了GDB会话，就不得不费时费力地重新设置这些内容。

现在，如何跟踪程序错误？在每次迭代结束时检查结果，是调试这个程序的一种自然方式。主要结果在“未完成”集合中（即在数组notdone[]中），以及在从0到其他顶点之间的已知距离的当前列表中（即在数组mind[]中）。例如，在第一次迭代以后，“未完成”集合应该由顶点2、3、4和5组成，在该迭代中选择了顶点1。

有了这些信息后，让我们应用一下确认原则，并在dowork()中for循环的每次迭代后检查notdone[]和mind[]。

必须谨慎地确定设置断点的确切位置。虽然第108行算法的主循环的末尾像是一个很自然的位置，但是实际上可能这并不是最佳位置，因为GDB针对每个线程都在那里停止。我们应选择在OpenMP的一个single块内放置断点，这样可以只对于一个线程停止。

因此，我们是从第二次迭代起在循环的开始位置来检查上一次迭代的结果。

```
(gdb) b 92 if step >= 1
Breakpoint 3 at 0x80490e3: file dijkstra.c, line 92.
(gdb) c
Continuing.
there are 4 threads

Breakpoint 3, __ompc_func_0 () at dijkstra.c:93
93              {  md = MAXINT;
```

让我们确认第一次迭代确实选择了要移出“未完成”集合的正确顶点（顶点1）。

```
(gdb) p mv
$1 = 0
```

然而，这个假设还没有得到确认。查看这段代码可以发现，我们在第100行上忘记了设置mv。我们将它修改为：

```
{  md = mymd; mv = mymv;  }
```

因此，我们再次重新编译和运行该程序。正如本节（以及本书的其他地方）前面提到的，运行程序时不退出GDB非常有帮助。虽然可以在另一个终端窗口中运行该程序，但是为了有点儿变化，让我们在这里采用另一种不同的方法。我们通过执行dis命令来暂时禁用断点，然后从GDB中重新编译程序，再使用ena重新启用断点。

```
(gdb) dis
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
`/debug/dij' has changed; re-reading symbols.
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1209026880 (LWP 11712)]
[New Thread -1209029728 (LWP 11740)]
[New Thread -1219519584 (LWP 11741)]
[New Thread -1230009440 (LWP 11742)]
there are 4 threads
graph weights:
```

5

```
0  3  6  17  15  13
3  0  15  6  12  9
6  15  0  1  2  7
17  6  1  0  10  19
15  12  2  10  0  3
13  9  7  19  3  0
minimum distances:
3
6
17
15
13

Program exited with code 06.
(gdb) ena
```

得到的仍然是错误的答案。让我们再次检查该断点处的内容。

```
(gdb) r
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1209014592 (LWP 11744)]
[New Thread -1209017440 (LWP 11772)]
[New Thread -1219507296 (LWP 11773)]
[New Thread -1229997152 (LWP 11774)]
there are 4 threads
[Switching to Thread -1209014592 (LWP 11744)]

Breakpoint 3, __ompc_func_0 () at dijkstra.c:93
93              {  md = MAXINT;
(gdb) p mv
$2 = 1
```

mv现在至少有正确的值。再检查mind[]。

```
(gdb) p *mind@6
$3 = {0, 3, 6, 17, 15, 13}
```

注意，因为通过malloc()动态地构造了mind[]数组，因此不能使用GDB的print命令的一般形式。我们使用了GDB的人工数组功能。

无论如何，mind[]都是错误的。例如，mind[3]应当是3+6=9，然而它是17。让我们检查更新mind[]的代码。

```
(gdb) b 107 if me == 1
Breakpoint 4 at 0x8049176: file dijkstra.c, line 107.
```

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1209039168 (LWP 11779)]
[New Thread -1209042016 (LWP 11807)]
[New Thread -1219531872 (LWP 11808)]
[New Thread -1230021728 (LWP 11809)]
there are 4 threads
[Switching to Thread -1230021728 (LWP 11809)]

Breakpoint 4, __ompc_func_0 () at dijkstra.c:107
107             updatemind(startv,endv);
```

首先，确认startv和endv具有有意义的值。

```
(gdb) p startv
$4 = 1
(gdb) p endv
$5 = 1
```

这个块的大小仅为1吗？让我们看一下。

```
(gdb) p chunk
$6 = 1
```

当检查了chunk的计算结果后，我们意识到需要等分nv的线程数量。后者的值为6，它不能被我们的线程数4整除。我们先记下以后要插入一些错误捕获代码，然后暂时将线程数减少为3。

同样，做这件事时不要退出GDB。虽然GDB继承了首次调用时的环境变量，但是也可以在GDB中修改或设置这些变量的值，我们在这里所做的事用如下代码表示。

```
(gdb) set environment OMP_NUM_THREADS = 3
```

现在再次运行程序。

```
(gdb) dis
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1208707392 (LWP 11819)]
[New Thread -1208710240 (LWP 11847)]
```

```
[New Thread -1219200096 (LWP 11848)]
there are 3 threads
graph weights:
0   3   6   17   15   13
3   0   15   6   12   9
6   15   0   1   2   7
17   6   1   0   10   19
15   12   2   10   0   3
13   9   7   19   3   0
minimum distances:
3
6
7
15
12

Program exited with code 06.
(gdb) ena
```

唉，得到的仍然是错误的答案！继续检查mind[]的更新进程。

```
(gdb) r
Starting program: /debug/dij 6 1
[Thread debugging using libthread_db enabled]
[New Thread -1208113472 (LWP 11851)]
[New Thread -1208116320 (LWP 11879)]
[New Thread -1218606176 (LWP 11880)]
there are 3 threads
[Switching to Thread -1218606176 (LWP 11880)]

Breakpoint 4, __ompc_func_0 () at dijkstra.c:107
107               updatemind(startv,endv);
(gdb) p startv
$7 = 2
(gdb) p endv
$8 = 3
```

好了，在me=1的情况下，得到了startv和endv的正确值。因此，进入函数：

```
(gdb) s
[Switching to Thread -1208113472 (LWP 11851)]

Breakpoint 3, __ompc_func_0 () at dijkstra.c:93
93               {  md = MAXINT;
(gdb) c
```

```
Continuing.
[Switching to Thread -1218606176 (LWP 11880)]
updatemind (s=2, e=3) at dijkstra.c:69
69        for (i = s; i <= e; i++)
```

注意，由于上下文在这些线程之间切换，因此我们没有立即进入`updatemind()`。现在检查`i=3`的情况。

```
(gdb) tb 71 if i == 3
Breakpoint 5 at 0x8048fb2: file dijkstra.c, line 71.
(gdb) c
Continuing.
updatemind (s=2, e=3) at dijkstra.c:71
71           if (mind[mv] + ohd[mv*nv+i] < mind[i])
```

照常，应用确认原则。

```
(gdb) p mv
$9 = 0
```

有一个大问题。前面介绍过，在第一次迭代中，`mv`的值为1。为什么这里得到的值是0呢？

回过神来，我们认识到环境切换应当跟它有关。再看一下上面的GDB输出。系统ID为11851的线程已经在第93行上；换言之，它已经在该算法主循环的下一次迭代中。事实上，当我们按下`c`来继续时，它甚至执行了第94行，即

```
mv = 0;
```

该线程重写了`mv`以前的值1，因此更新`mind[3]`的线程现在依赖于`mv`的错误值。解决方法是添加另一个屏障（barrier）。

```
updatemind(startv,endv);
#pragma omp barrier
```

修复这个问题后，程序就可以正确地运行。

前述代码是用Omni编译器的。正如我们所提到的，从版本4.2起，GCC也能处理OpenMP代码了。只要在GCC命令行上添加`-fopenmp`标记即可。

与Omni不同的是，GCC生成代码时，从一开始，GDB的焦点就在你自己的源代码中。因此，在GDB会话的开端执行如下命令。

```
(gdb) b main
```

实际上会导致在自己的`main()`中设置一个断点，这与我们在Omni编译器中看到的情况不同。

然而，在编写本书时，GCC的一个主要缺点是：OpenMP `parallel`块内的局部变量（OpenMP

术语称之为私有变量）的符号在GDB中不可见。例如，对上面Omni生成的代码执行如下命令对GCC生成的代码也适用：

```
(gdb) p mv
```

但是命令

```
(gdb) p startv
```

则不适合于GCC生成的代码。

当然，这个问题有解决的办法。例如，要知道`startv`的值，可以查询`updatemind()`中`s`的值。但愿这个问题在GCC的下一个版本中会得到解决。

第 6 章

特殊主题

6

在调试过程中会产生各种调试工具没法处理的问题，本章将讨论其中的部分问题。

6.1 根本无法编译或加载

尽管GDB、DDD和Eclipse的威力不小，但是如果程序根本不能编译，调试工具再强大也无可奈何。本节介绍处理这种情况的一些技巧。

6.1.1 语法错误消息中的“幽灵”行号

有时编译器指出第*x*行有语法错误，而事实上第*x*行完全正确，真正的错误在前面某一行上。

例如，下面是第3章的源文件*bintree.x*，在还没有指出的某个地方抛出了一个语法错误（其实如果要查找这个错误，是相当明显的）。

```
1  //  bintree.c:  routines to do insert and sorted print of a binary tree
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  struct node {
7     int val;              // stored value
8     struct node *left;    // ptr to smaller child
9     struct node *right;   // ptr to larger child
10 };
11
12 typedef struct node *nsp;
13
14 nsp root;
15
16 nsp makenode(int x)
17 {
18    nsp tmp;
19
```

```
20    tmp = (nsp) malloc(sizeof(struct node));
21    tmp->val = x;
22    tmp->left = tmp->right = 0;
23    return tmp;
24 }
25
26 void insert(nsp *btp, int x)
27 {
28    nsp tmp = *btp;
29
30    if (*btp == 0) {
31       *btp = makenode(x);
32       return;
33    }
34
35    while (1)
36    {
37       if (x < tmp->val) {
38
39          if (tmp->left != 0) {
40             tmp = tmp->left;
41          } else {
42             tmp->left = makenode(x);
43             break;
44          }
45
46       } else {
47
48          if (tmp->right != 0) {
49             tmp = tmp->right;
50          } else {
51             tmp->right = makenode(x);
52             break;
53          }
54
55    }
56 }
57
58 void printtree(nsp bt)
59 {
60    if (bt == 0) return;
61    printtree(bt->left);
62    printf("%d\n",bt->val);
63    printtree(bt->right);
64 }
```

```
65
66  int main(int argc, char *argv[])
67  {
68     int i;
69
70     root = 0;
71     for (i = 1; i < argc; i++)
72        insert(&root, atoi(argv[i]));
73     printtree(root);
74  }
```

通过GCC运行这段代码产生的结果为：

```
$ gcc -g bintree.c
bintree.c: In function `insert':
bintree.c:75: parse error at end of input
```

由于第74行是源文件的末尾，因此第二条错误消息至少可以说提供的信息量相当少。但是第一条消息表明问题在`insert()`中，因此这是一个线索，尽管这条消息没有指出问题是什么，也没有指出问题在何处。

在这种情况下，典型的出错原因是少了闭合花括号或分号。可以直接检查有没有缺少这样的符号，但是在大文件中这样做比较困难。下面我们采用另一种方法。

第1章介绍过确认原则。这里，先确认问题确实在`insert()`中。要进行这样的确认，暂时先将该函数从源代码中注释掉。

```
...
   tmp->val = x;
   tmp->left = tmp->right = 0;
   return tmp;
}

// void insert(nsp *btp, int x)
// {
//    nsp tmp = *btp;
//
//    if (*btp == 0) {
//       *btp = makenode(x);
//       return;
//    }
//
//    while (1)
//    {
//       if (x < tmp->val) {
//
//          if (tmp->left != 0) {
//             tmp = tmp->left;
```

6

```
//          } else {
//             tmp->left = makenode(x);
//             break;
//          }
//
//       } else {
//
//          if (tmp->right != 0) {
//             tmp = tmp->right;
//          } else {
//             tmp->right = makenode(x);
//             break;
//          }
//
//    }
// }

void printtree(nsp bt)
{
   if (bt == 0) return;
...
```

说明　最好使用快捷方式来注释掉该函数，比如采用块操作。第7章将介绍适用于调试环境的文本编辑器快捷方式。

保存文件，然后重新运行GCC。

```
$ gcc -g bintree.c
/tmp/ccg0LDCS.o: In function `main':
/home/matloff/public_html/matloff/public_html/Debug/Book/DDD/bintree.c:72:
undefined reference to `insert'
collect2: ld returned 1 exit status
```

不要被连接器LD关于找不到`insert()`的抱怨分散注意力。毕竟早就知道肯定会有这样的抱怨，因为前面已经注释掉了该函数。重要的是已经没有像以前那样抱怨语法错误。因此，确认了语法错误在`insert()`中。现在，取消该函数的注释（同样，最好使用文本编辑器快捷方式，比如“撤销”）并保存文件。另外，为了确保已正确地恢复，重新运行GCC以确认这一语法错误再次出现（这里不再显示该错误）。

这时可以应用第1章介绍的另一个原则：二分搜索原则。反复缩小函数`insert()`中的搜索区域，每次将搜索区域切成两半，直到获得足够小的区域找出语法错误。

为了达到这个目的，首先将该函数的大约一半内容注释掉。做这件事的一种合理方式是直接将`while`循环注释了，然后重新运行GCC。

```
$ gcc -g bintree.c
$
```

这时发现错误消息消失了，因此语法问题肯定在这个循环中的某个地方。那么，已将问题的范围缩小到了该函数的一半，现在继续将这一区域切成两半。为此，将`else`代码也注释掉。

```
void insert(nsp *btp, int x)
{
   nsp tmp = *btp;

   if (*btp == 0) {
      *btp = makenode(x);
      return;
   }

   while (1)
   {
      if (x < tmp->val) {

         if (tmp->left != 0) {
            tmp = tmp->left;
         } else {
            tmp->left = makenode(x);
            break;
         }

       } // else {
//
//          if (tmp->right != 0) {
//             tmp = tmp->right;
//          } else {
//             tmp->right = makenode(x);
//             break;
//          }
//
//    }
}
```

重新运行GCC，发现问题再次出现：

```
$ gcc -g bintree.c
bintree.c: In function `insert':
bintree.c:75: parse error at end of input
```

因此，语法错误要么在`if`块中，要么在函数末尾。这时，已经将问题范围缩小到了只有7行的代码内，因此直接通过观察应该就可以查出问题：原因在于不小心漏掉了外部`if-then-else`

中的闭合花括号。

二分搜索原则对于查找未知位置的语法错误非常有帮助。但是在暂时注释掉代码的过程中，一定不要造成新的语法错误！要像我们这样，将整个函数、整个循环一起注释掉。

6.1.2　缺少库

有时GCC（实际上是构建程序的过程中GCC调用的连接器LD）会通知你无法找到代码调用的一个或多个函数。这通常是由于没有成功地将库函数的位置通知GCC。本书的很多读者肯定都精通这一主题，但是为了照顾不精通这一主题的读者，我们在本节进行一下简单的介绍。注意，本节的讨论主要适用于Linux，它也基本适用于其他Unix类型的操作系统。

1. 示例

让我们使用下面的简单代码作为示例，它包含*a.c*中的一个主程序：

```
// a.c

int f(int x);

main()
{
   int v;
   scanf("%d",&v);
   printf("%d\n",f(v));
}
```

以及*z/b.c*中的一个子程序：

```
// b.c

int f(int x)
{
   return x*x;
}
```

如果试图在没有连接到*b.c*中的代码时就编译*a.c*，那么LD当然会抱怨。

```
$ gcc -g a.c
/tmp/ccIP5WHu.o: In function `main':
/debug/a.c:9: undefined reference to `f'
collect2: ld returned 1 exit status
```

我们可以进入*z*中，编译*b.c*，然后连接目标文件。

```
$ cd z
$ gcc -g -c b.c
$ cd ..
$ gcc -g a.c z/b.o
```

然而，如果有很多函数要连接，这些函数可能来自不同的源文件，而且这些函数对于将来要编写的程序可能有用，那么可以创建一个库（这是一个存档文件）。库文件分为两种。当编译调用静态库中函数的代码时，那些函数变成最终可执行文件的一部分。另一方面，如果库是动态的，那么只有实际执行了程序，这些函数才会真正附加到调用代码上。

下面介绍如何为本节的示例创建静态库，假设名为*lib88.a*。

```
$ gcc -g -c b.c
$ ar rc lib88.a b.o
```

这里的`ar`命令从它在文件*b.o*找到的函数中创建了库*lib88.a*。然后编译主程序：

```
$ gcc -g a.c -l88 -Lz
```

这里的`-l`选项是一种快捷方式，与如下代码的效果相同：

```
$ gcc -g a.c lib88.a -Lz
```

这个选项指示GCC告诉LD，它将需要在库*lib88.a*（或者下面将要介绍的动态变体）中查找函数。

说明 在Unix系统上，按惯例是在静态库文件名后面加上后缀*.a*，*a*代表`archive`。另外，任何库的名称一般都以`lib`开头。

`-L`选项指示GCC告诉LD在查找函数时顺便在其他目录中也查找一下（除当前目录和默认搜索目录以外）。在本节的示例中，*z*就是这样的目录。

这种方法的缺点是，如果有很多程序在使用同一个库，那么每个程序都会在磁盘上包含该库的独立副本，这样比较浪费空间。使用动态库可以解决这个问题（代价是需要一点额外的加载时间）。

在这里的示例中，使用GCC直接创建动态库，而不是使用*ar*。在*z*中将运行：

```
$ gcc -fPIC -c b.c
$ gcc -shared -o lib88.so b.o
```

这段代码创建了动态库*lib88.so*。（Unix中命名动态库的惯例是使用后缀*.so*，表示*shared object*，后面可能会跟着版本号。）与连接静态库一样地连接到这个动态库：

```
$ gcc -g a.c -l88 -Lz
```

然而，它现在的工作方式有所不同。在静态库的情况下，从库中调用的函数会成为可执行文件*a.out*的一部分，现在*a.out*仅包含一个表示程序使用了库*lib88.so*的符号。重要的是，这种符号甚至没有表明该库位于何处。GCC（同样，实际上是LD）要在编译时查看*lib88.so*的唯一原因是为了得到连接所需知道的该库的信息。

连接本身会在运行时发生。操作系统会搜索*lib88.so*，然后将它连接到程序中。这就带来了操作系统在何处执行这种搜索的问题。

首先，使用`ldd`命令检查程序需要哪些库，如果有，操作系统可以在何处找到它们。

```
$ ldd a.out
        lib88.so => not found
        libc.so.6 => /lib/tls/libc.so.6 (0x006cd000)
        /lib/ld-linux.so.2 (0x006b0000)
```

该程序需要C库，并在目录*/lib/tls*中找到了它，但是操作系统没有找到*lib88.so*。后者在目录*/Debug/z*中，但是该目录不是操作系统的正常搜索路径的一部分。

解决这种问题的一种方式是向该搜索路径中添加*/Debug/z*：

```
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/Debug/z
```

如果要添加几个目录，将目录名连同冒号的字符串作为分隔符。（这适用于C shell或TC shell。）对于`bash`，执行如下命令：

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Debug/z
$ export LD_LIBRARY_PATH
```

让我们确保它能成功运行：

```
$ ldd a.out
        lib88.so => /Debug/z/lib88.so (0xf6ffe000)
        libc.so.6 => /lib/tls/libc.so.6 (0x006cd000)
        /lib/ld-linux.so.2 (0x006b0000)
```

虽然还有其他多种方法，但是那些方法不在本书介绍范围之列。

2. 开源软件中库的用法

开源软件现在很流行，尤其是在Linux用户之间。然而有时会出现一个问题，即与源代码配套的构建脚本（通常起名为*configure*）找不到某些必需的库。试图通过设置`LD_LIBRARY_PATH`环境变量解决该问题可能会失败。由于这个问题处于这里讨论的“缺少库”主题下，而且源代码包中通常没有记载，因此这里有必要做一下简单的解释。

这种问题的根源常常在于*configure*调用的名为*pkgconfig*的程序。这个程序会从某些元数据文件处检索关于库的信息。这样的文件后缀为*.pc*，前缀是库的名称。例如，文件*libgcj.pc*中包含库文件*libgcj.so**的位置。

*pkconfig*搜索*.pc*文件的默认目录取决于*pkgconfig*本身的位置。例如，如果程序位于*/usr/bin*中，则搜索*/usr/lib*。如果所需的库是*/usr/local/lib*，仅在这个目录中搜索就不够。为了解决这个问题，应设置环境变量为`PKG_CONFIG_PATH`。在C或TC shell中，执行如下`shell`命令：

```
% setenv PKG_CONFIG_PATH /usr/lib/pkgconfig:/usr/local/lib/pkgconfig
```

6.2　调试GUI程序

如今，用户习惯于应用程序带有GUI（图形用户界面）。当然，GUI也是程序，因此可以应用

一般调试原理，但是会有一些特殊考虑事项。

GUI编程主要是调用某个库以便在屏幕上执行各种操作。市面上有很多很多这样的库，用途相当广泛。显然我们无法一一覆盖，但是不管是什么库，原理都是相似的。

因此，我们选择了最简单的示例，curses库。这个库简单到很多人可能根本没有把它当成GUI（有个学生称之为“基于文本的GUI”），但是它能说明问题。

调试curses程序

程序员可以使用curses库编写让光标在屏幕上移动的代码，改变字符的颜色，或者改成反白显示，插入及删除文本，等等。

例如，像Vim和Emacs这样的文本编辑器就是用curses编写的。在Vim中，按下`j`键使光标向下移一行。键入`dd`导致删除当前行，这一行下面的代码行向上移一行，上面的代码行仍然不变。这些动作通过调用curses库中的函数实现。

为了使用curses，必须在源代码中包括如下语句：

```
#include <curses.h>
```

还必须连接到curses库：

```
gcc -g sourcefile.c -lcurses
```

以下面这段代码为例。该代码运行Unix的命令`ps ax`来列出所有进程。在任何给定时间，都必须突出显示光标当前所在的行。按下`u`和`d`键可以将光标向上或向下移动，等等。参见代码中的注释查看完整的命令列表。

如果以前没有使用过curses，不要担心，因为注释中说明了这个库是做什么的。

```
// psax.c; illustration of curses library

// read this code in a "top-down" manner:  first these comments and the global
// variables, then main(), then the functions called by main()

// runs the shell command 'ps ax' and saves the last lines of its output,
// as many as the window will fit; allows the user to move up and down

// within the window, with the option to kill whichever process is
// currently highlighted

// usage:  psax

// user commands:

//    'u':  move highlight up a line
//    'd':  move highlight down a line
//    'k':  kill process in currently highlighted line
```

```
//    'r':  re-run 'ps ax' for update
//    'q':  quit

// possible extensions:  allowing scrolling, so that the user could go
// through all the 'ps ax' output, not just the last lines; allow
// wraparound for long lines; ask user to confirm before killing a
// process

#define MAXROW 1000
#define MAXCOL 500

#include <curses.h>  // required

WINDOW *scrn; // will point to curses window object

char cmdoutlines[MAXROW][MAXCOL];  // output of 'ps ax' (better to use
                                   // malloc())
int ncmdlines,  // number of rows in cmdoutlines
    nwinlines,  // number of rows our "ps ax" output occupies in the
                //  xterm (or equiv.) window
    winrow,  // current row position in screen
    cmdstartrow,  // index of first row in cmdoutlines to be displayed
    cmdlastrow;  // index of last row in cmdoutlines to be displayed

// rewrites the line at winrow in bold font
highlight()
{
   int clinenum;
   attron(A_BOLD);  // this curses library call says that whatever we
                    // write from now on (until we say otherwise)
                    // will be in bold font
   // we'll need to rewrite the cmdoutlines line currently displayed
   // at line winrow in the screen, so as to get the bold font
   clinenum = cmdstartrow + winrow;
   mvaddstr(winrow,0,cmdoutlines[clinenum]);
   attroff(A_BOLD);  // OK, leave bold mode
   refresh();  // make the change appear on the screen
}

// runs "ps ax" and stores the output in cmdoutlines
runpsax()
{
   FILE *p; char ln[MAXCOL]; int row,tmp;
   p = popen("ps ax","r");  // open UNIX pipe (enables one program to read
                            // output of another as if it were a file)
   for (row = 0; row < MAXROW; row++)  {
```

```
      tmp = fgets(ln,MAXCOL,p);  // read one line from the pipe
      if (tmp == NULL) break;  // if end of pipe, break
      // don't want stored line to exceed width of screen, which the
      // curses library provides to us in the variable COLS, so truncate
      // to at most COLS characters
      strncpy(cmdoutlines[row],ln,COLS);
      cmdoutlines[row][MAXCOL-1] = 0;
   }
   ncmdlines = row;
   close(p);  // close pipe
}

// displays last part of command output (as much as fits in screen)
showlastpart()
{
   int row;
   clear();  // curses clear-screen call
   // prepare to paint the (last part of the) 'ps ax' output on the screen;
   // two cases, depending on whether there is more output than screen rows;
   // first, the case in which the entire output fits in one screen:
   if (ncmdlines <= LINES)  { // LINES is an int maintained by the curses
                              // library, equal to the number of lines in
                              // the screen
      cmdstartrow = 0;
      nwinlines = ncmdlines;
   }
   else  { // now the case in which the output is bigger than one screen
      cmdstartrow = ncmdlines - LINES;
      nwinlines = LINES;
   }
   cmdlastrow = cmdstartrow + nwinlines - 1;
   // now paint the rows to the screen
   for (row = cmdstartrow, winrow = 0; row <= cmdlastrow; row++,winrow++)
      mvaddstr(winrow,0,cmdoutlines[row]);  // curses call to move to the
                                            // specified position and
                                            // paint a string there
   refresh();  // now make the changes actually appear on the screen,
               // using this call to the curses library
   // highlight the last line
   winrow--;
   highlight();
}

// moves cursor up/down one line
updown(int inc)
{
```

```
   int tmp = winrow + inc;
   // ignore attempts to go off the edge of the screen
   if (tmp >= 0 && tmp < LINES)  {
      // rewrite the current line before moving; since our current font
      // is non-BOLD (actually A_NORMAL), the effect is to unhighlight
      // this line
      mvaddstr(winrow,0,cmdoutlines[winrow]);
      // highlight the line we're moving to
      winrow = tmp;
      highlight();
   }
}

// run/re-run "ps ax"
rerun()
{
   runpsax();
   showlastpart();
}

// kills the highlighted process
prockill()
{
   char *pid;
   // strtok() is from C library; see man page
   pid = strtok(cmdoutlines[cmdstartrow+winrow]," ");
   kill(atoi(pid),9);  // this is a UNIX system call to send signal 9,
                       // the kill signal, to the given process
   rerun();
}

main()
{
   char c;
   // window setup; next 3 lines are curses library calls, a standard
   // initializing sequence for curses programs
   scrn = initscr();
   noecho();  // don't echo keystrokes
   cbreak();  // keyboard input valid immediately, not after hit Enter
   // run 'ps ax' and process the output
   runpsax();
   // display in the window
   showlastpart();
   // user command loop
   while (1)  {
      // get user command
```

```
        c = getch();
        if (c == 'u') updown(-1);
        else if (c == 'd') updown(1);
        else if (c == 'r') rerun();
        else if (c == 'k') prockill();
        else break;  // quit
    }
    // restore original settings
    endwin();
}
```

运行这个程序，将发现从图上看一切正常，但是当按下u键使光标向上移动一行时，却没有正确地反应，如图6-1所示。

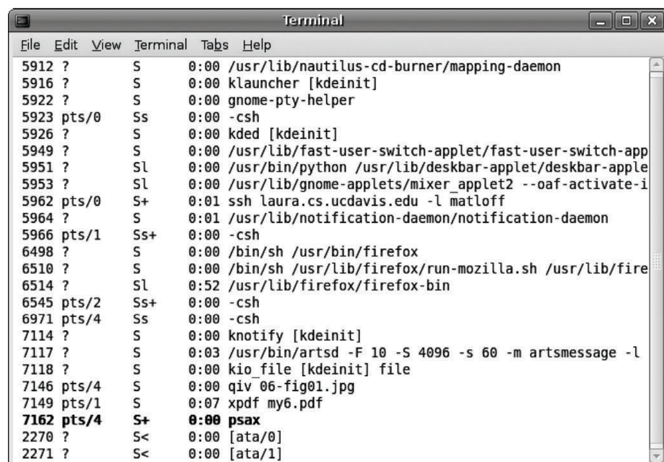


图6-1　某个终端窗口中的程序运行情况

`ps ax`的输出是按进程号的升序排列的，然而突然发现2270显示在7162后面。让我们跟踪这个程序错误。

curses程序是编写调试类图书作者的一个梦想，因为它使程序员不得不使用调试工具。程序员不能使用`printf()`调用或`cout`语句来输出调试信息，因为这样的输出会与程序输出本身混合在一起，从而造成混乱。

1. 使用GDB

启动GDB，但是相对于上一次，我们必须做一件额外的事。我们必须告诉GDB让程序在不同的终端窗口中执行，而不是GDB正在其中运行的那个窗口。可以使用GDB的`tty`命令来完成这件事。首先，进入另一个用来完成程序I/O的窗口，并在其中运行Unix的`tty`命令来确定该窗口的ID。在这种情况下，该命令的输出告诉我们，那个窗口是终端号`/dev/pts/8`，因此在GDB窗口中输入：

```
(gdb) tty /dev/pts/8
```

从现在起，该程序的所有键盘输入和屏幕输出将位于执行窗口中。

在开始前的最后一件事是：必须在执行窗口中键入类似如下代码：

```
sleep 10000
```

这样在该窗口中的键盘输入会进入程序，而不是进入shell。

说明　还有其他方式可以处理将GDB输出与程序输出分开的问题。例如，可以先开始程序的执行，然后在另一个窗口中激活GDB，将它附加到正在运行的程序后面。

接下来，因为错误发生在试图将光标向上移的时候，所以在函数`updown()`的开头设置一个断点。然后，当键入`run`时，程序会开始在执行窗口中执行。在该窗口中按下u键，GDB会在设置的断点处停止。

```
(gdb) r
Starting program: /Debug/psax
Detaching after fork from child process 3840.

Breakpoint 1, updown (inc=-1) at psax.c:103
103     {  int tmp = winrow + inc;
```

首先，确认变量`tmp`值的正确性。

```
(gdb) n
105         if (tmp >= 0 && tmp < LINES)  {
(gdb) p tmp
$2 = 22
(gdb) p LINES
$3 = 24
```

变量`winrow`显示了光标在窗口中的当前位置。那个位置应该就在窗口末尾。`LINES`的值为24，因此`winrow`应当为23，因为是从0开始编号的。`inc`等于−1（因为我们在将光标向上移，而不是向下移），所以如这里所示，确认了`tmp`的值为22。

现在，让我们来到下一行。

```
(gdb) n
109             mvaddstr(winrow,0,cmdoutlines[winrow]);
(gdb) p cmdoutlines[winrow]
$4 = " 2270 ?        Ss      0:00 nifd -n\n", '\0' <repeats 464 times>
```

显然，这是来到了进程2270的那一行。我们很快意识到源代码中的如下代码行：

```
mvaddstr(winrow,0,cmdoutlines[winrow]);
```

应当为：

```
mvaddstr(winrow,0,cmdoutlines[cmdstartrow+winrow]);
```

修复了这个问题后，程序就能正确地运行。

完成后，在执行窗口中按下Ctrl+C组合键，以便终止`sleep`命令，并使shell再次回到可用状态。

注意，如果因为出了错使程序过早结束，那么该执行窗口可能会保留部分非标准终端设置，例如，cbreak模式。为了修复这一问题，来到那个窗口中并按下Ctrl+J 组合键，然后键入单词`reset`，之后再次按下Ctrl+J组合键。

2. 使用DDD

使用DDD调试是什么情况呢？同样，需要一个单独的窗口来执行程序。选择View→Execution Window，DDD会弹出一个执行窗口。注意，不需要亲自在该窗口中键入`sleep`命令，因为DDD替你做了这件事。屏幕显示将如图6-2所示。

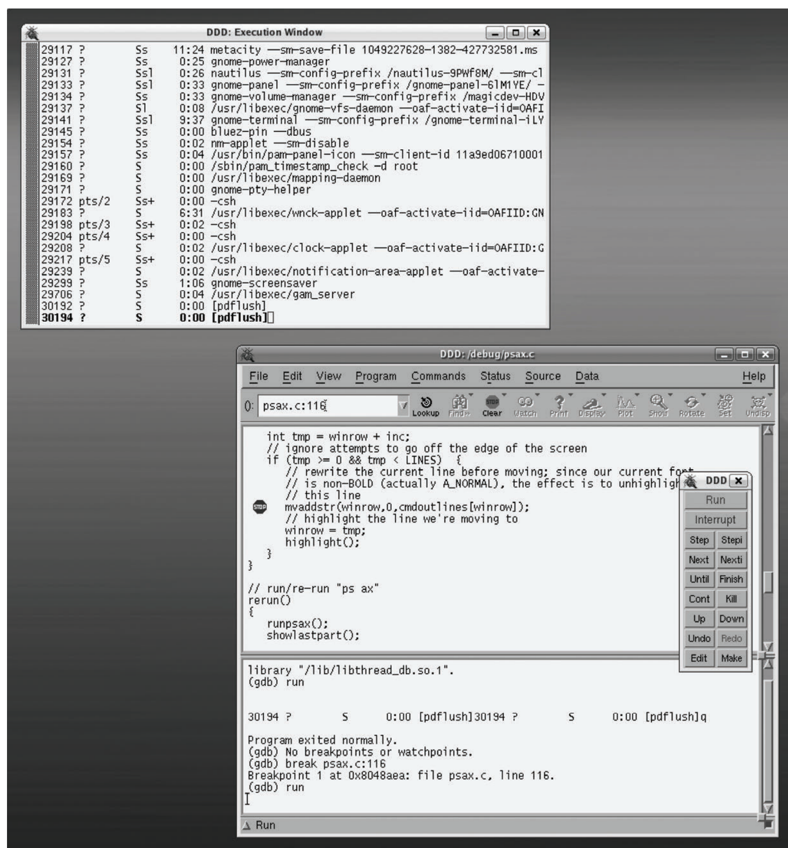


图6-2　在DDD中附加到运行程序上

照样设置断点，但是记住要在执行窗口中键入程序输入。

3. 使用Eclipse

首先要注意，在构建项目时，需要让Eclipse在make文件中使用`-lcurses`标志，这一过程参见第5章的相关介绍。

这里也需要一个单独的执行窗口。可以在建立调试对话框时做这件事。当像惯常一样建立运行对话框并选择Run→Open Debug Dialog时，我们将采取与目前所采用方法略微不同的其他方式。在图6-3所示界面中，注意到除了通常的选项C/C++ Local Application外，还有选项C/C++ Attach to Local Application。后者意味着要让Eclipse使用GDB能力来将它本身附加到一个已经运行的进程上（第5章讨论过）。右击C/C++ Attach to Local Application，选择New，并像以前一样继续进行下去。

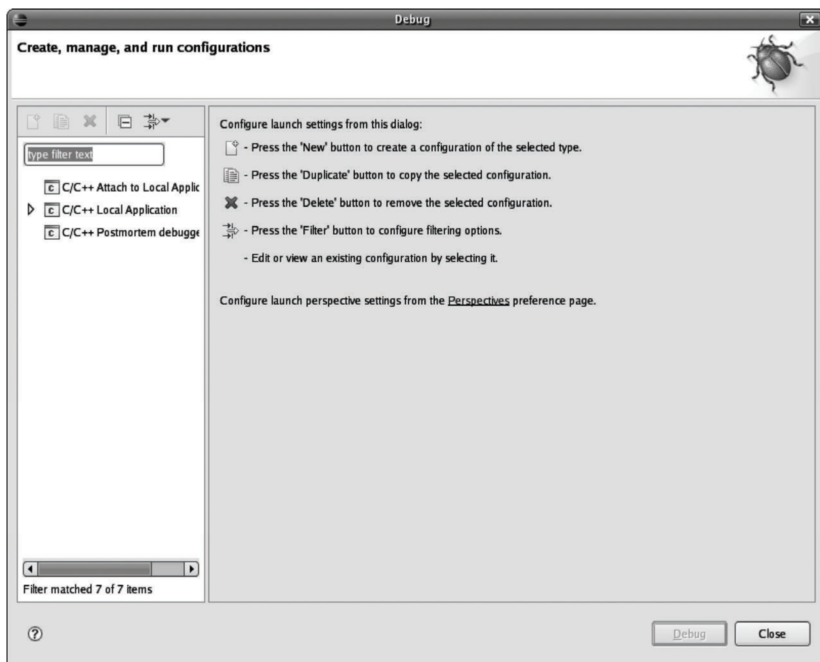


图6-3　在Eclipse中附加到运行程序上

当开始实际运行调试时，首先在一个单独的shell窗口中启动程序。（不要忘记，该程序可能位于Eclipse工作空间目录中。）然后像首次进行调试运行时常做的那样，选择Run→Open Debug Dialog。在这种情况下，Eclipse会弹出一个窗口列出所有进程，要求你选择希望GDB附加到哪个进程。这一过程如图6-4所示，其中显示了`psax`进程的ID为12319（注意，该程序在另一个窗口中运行，这里局部地隐藏了）。单击该进程，然后单击OK，出现图6-5所示的情形。

在图6-5中，可以看到程序在系统调用期间停止了。Eclipse发出通知，表明它没有当前指令的源代码，但这是预料之中的，没有问题。实际上，这是在源文件*psax.c*中设置断点的好时机。设置以后，单击Resume图标。Eclipse会一直运行，直到遇到第一个断点，然后可以照例调试。

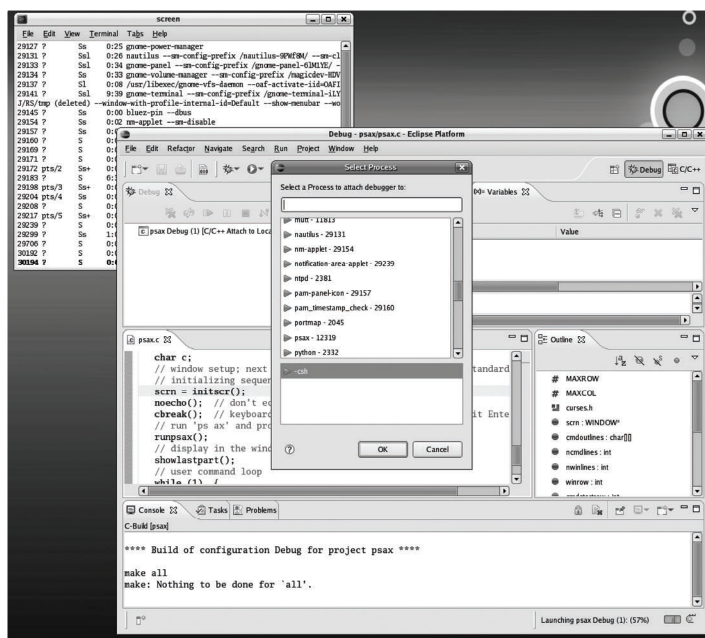


图6-4 选择要将GDB附加到的进程

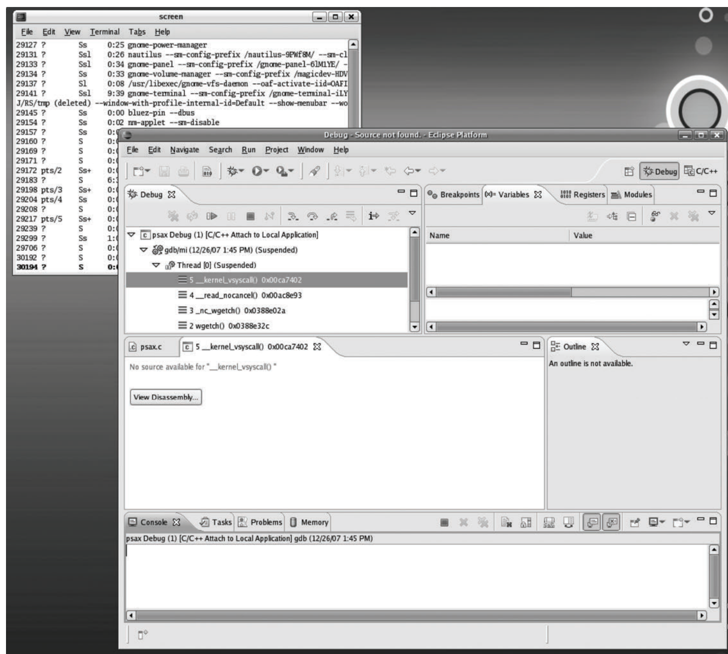


图6-5 在内核中停止

第7章

其他工具

精通调试代码并不是说学会使用GDB这样的调试器就行了——这只是开端。市面上有各种各样的其他调试工具，免费的和需付费的都有，它们也可以帮助防止、检测和消除代码中的程序错误。为了增强调试技能，初学者程序员最好学会其中的几种调试工具，了解哪种工具适合于调试哪种程序错误，并识别发生程序错误时使用其中哪个工具可以节省时间和精力。

到目前为止我们的焦点都集中在使用符号调试器上，现在我们把覆盖面扩展到调试的其他方面，包括防御性编程。本章专门介绍除GDB外的部分工具和技术，这些工具和技术对于在一开始防止程序错误产生以及程序错误产生后进行查找及修复都是很有用的。

7.1 充分利用文本编辑器

最好的调试方法是一开始就不要有编程错误！人们往往最容易忽略一种“预调试”方式：充分用好一种支持编程的编辑器。

如果你花在编码上的时间很长，我们强烈建议你仔细考虑选择哪种编辑器，并尽可能地充分学习将使用的编辑器。这样做有两个原因。首先，精通强大的编辑器可以缩短编写代码所需的时间。具有自动缩排、单词补全和全文符号查询等特殊功能的编辑器对程序员非常有利，市面上现在有数种这样的编辑器可以选择。其次，优秀编辑器确实可以帮助编码者在编写代码时捕获某些类型的程序错误，这正是本节要介绍的内容。

本书两位作者都是使用Vim编程的，因此这种编辑器是我们将重点介绍的内容。然而，所有流行编辑器都有相似（即便不相同）的功能。如果Vim能提供Emacs中目前没有的功能，那么使用Emacs的开发人员会很快转而使用Vim，反之亦然。因此，虽然我们特指Vim，但是这里介绍的大部分内容也适用于Emacs等其他优秀编辑器。

7.1.1 语法突出显示

Vim的语法突出显示功能可以用不同颜色或字体显示程序的一部分，因此像关键字、类型标识符、局部变量和预处理器指令等代码元素都有各自的颜色和字体模式。编辑器通过查看文件名的扩展名来判断使用的是什么语言，然后选择相应的颜色和字体。例如，如果文件名以*.pl*结尾（表

示一个Perl脚本），那么突出显示单词*die*（它是Perl函数名），而如果以*.c*结尾，就不突出显示这个单词。

更好的语法突出显示的名称是词汇突出显示，因为编辑器一般不会太细致地分析语法。它不能指出在函数调用中参数数量有错或者参数类型有误。它只能理解*bless*和*foreach*等单词是Perl关键字，*fmt*和*dimension*是Fortran关键字，并相应地显示它们。

即便如此，语法突出显示对于捕获简单但容易犯的错误仍然非常有用。例如，在我们的计算机上，像C语言中的`FILE`或`float`关键字这样的类型标识符，默认颜色是绿色。当眼睛习惯于这种字体和颜色后，无需经过不必要的编译周期，误拼时就可以发现颜色不一致，并自动纠正这一错误。

使用语法突出显示的一个示例是检查出现在make文件中的我们（作者）喜欢的关键字。`patsubst`关键字是一种用于make文件的非常有用的文本查找和替换命令。它的一个最常见的用法是生成项目源代码*.c*文件中的*.o*文件列表。

```
TARGET = CoolApplication
OBJS = $(patsubst %.c, %.o, $(wildcard *.c))

$(TARGET): $(OBJS)
```

本书的作者中有一位始终不记得它是`patsubst`、`pathsubst`还是`patsub`。知道了make文件关键字是用亮色（黄色）显示后，你能看出图7-1所示代码行中哪个版本是错误的吗？即使你不知道如何编写make文件，仅仅依靠突出显示就可以一目了然！[①]

```
OBJS = $(patsub %.c, %.o $(wildcard *.c))
OBJS = $(patsubst %.c, %.o $(wildcard *.c))
```

图7-1 语法突出显示表明我的make文件是错误的

而且，下面是一个更为灵活的语法突出显示示例。图7-2中有一个实实在在的语法错误。这时，不需要过多地思考错误是什么（或错误在何处），只要根据颜色即可找出错误。

```
if (fp == NULL) {
    puts("This was a "bad" file pointer.");
    exit(1);
}
```

图7-2 语法突出显示揭示了一个常见的错误

图7-3是另一个类似错误的图示。试着根据颜色找出错误。

① 本书将此图转换成了灰度模式，在实际应用中用颜色区别更容易识别错误。

```
fprintf(fp, "argument %d is \"%s\".\n, i, argv[i]);
printf("I just wrote \"%s\" which is argument %d\n", argv[i], i);
```

图7-3　语法突出显示揭示了另一个常见错误

如果发现语法突出显示模式中的某些颜色难以阅读，可以通过键入如下代码关闭突出显示：

```
: syntax off
```

再次打开颜色模式的命令当然是：

```
: syntax on
```

更好的办法是修改语法文件，对那种类型的关键字使用另一种更好的颜色，但是这种办法超出了我们这里的讨论范围。

7.1.2　匹配括号

说明　本节中的括号是指圆括号、方括号和花括号，即()、[]和{}。

括号不对称错误极其常见，且难以捕获。来看如下代码。

```
mytype *myvar;
if ((myvar = (mytype *)malloc(sizeof(mytype))) == NULL) {
   exit(-1);
}
```

请立即回答：代码中圆括号是否对称？[①]你是否曾不得不在带大量条件语句的代码块中跟踪不对称的括号？或者有没有用过TEX？（我们为过去丢失了括号的LATEX文件而汗颜！）如果有，那么你肯定会同意我们的观点：这完全是计算机要负责的事情，怎能让我们去做这么繁琐的工作！Vim有一些不错的功能有助于检查括号是否匹配。

- 每当在键盘上键入括号时，Vim的showmathc选项使Vim暂时把光标放在匹配的括号上（如果匹配括号存在并且在屏幕上可见）。通过设置matchtime变量，甚至可以控制光标在匹配括号上停留多长时间（以十分之一秒为单位）。
- 当光标在一个括号上时，键入百分号会将光标移到配对括号上。这个命令是跟踪不对称括号的一个极佳方法。
- 将光标放在一个括号上时，Vim会突出显示与其匹配的另外一个括号，如图7-2所示。

编程时showmatch选项是有用的，但是其他时候这种特性比较讨厌。可以使用自动命令来设置这个选项仅在编程时生效。例如，为了设置showmatch仅适用于C/C++源代码文件的编辑会话，可以将类似如下代码行放在*.vimrc*文件中（参见Vim的帮助文件了解更多信息）。

---

① 你还是老老实实承认吧，你一定被我唬住了，还以为它们是不对称的。我们的重点不是指出你错了，而在于说明要找出答案是要花一些时间的。

```
au BufNewFile,BufRead *.c set showmatch
au BufNewFile,BufRead *.cc set showmatch
```

如果代码中有不对称的括号，或者更糟糕的情况，需要检查别人的杂乱无章的程序代码中的不对称括号，该怎么办呢？前面提到的%编辑器命令可以搜索对称的成组字符。例如，如果将光标放在左方括号[上，并从命令模式下键入%，Vim就会将光标的位置放到下一个]字符上。如果将光标放在右花括号}上，并调用%，那么Vim会将光标放在前一个匹配的{字符上。通过这种方式，不仅可以验证任何给定的圆括号、花括号或方括号有对应的匹配括号，而且可以验证匹配的括号符合语义。使用Vim的matchpair命令甚至可以定义其他匹配的“括号”对，比如HTML的注释定界符<!--和-->。参见Vim的帮助页面可了解更多信息。

7.1.3　Vim与make文件

make实用工具管理Linux/Unix系统上可执行文件的编译和构建，程序员花一点点精力学习它的用法可以得到巨大的回报。然而，这也引入了新的出错机会。如果使用make，Vim有几个有益于调试过程的功能。来看下面的make文件片段。

```
all: yerror.o main.o
	gcc -o myprogram yerror.o main.o

yerror.o: yerror.c yerror.h
	gcc -c yerror.c

main.o: main.c main.h
   gcc -c main.c
```

该make文件中有一个错误，但是很难看出来。make对于格式非常挑剔。目标的命令行必须以一个制表符开头，而不是以空格开头。只要从Vim中执行set list命令，马上就可以发现错误所在。

```
all: yerror.o main.o$
^Igcc -o myprogram yerror.o main.o$
$
yerror.o: yerror.c yerror.h$
^Igcc -c yerror.c$
$
main.o: main.c main.h$
   gcc -c main.c$
```

在list模式中，Vim显示不可打印的字符。默认情况下，行末的字符显示为$，控制字符显示为插入符号（^），因此，制表符（Ctrl+I）显示为^I。因而可以将空格与制表符区分开，错误很容易看出：*main.o* make目标的命令行是以空格开头的。

使用Vim的listchars选项可以控制显示内容。例如，如果要将行末字符改成=而不是$，可以使用:set listchars=eol:=。

7.1.4　make文件和编译器警告

从Vim中调用make非常方便。例如，不需要手工保存文件并在另一个窗口中键入make clean，只要从命令模式下键入:make clean即可。（一定要设置autowrite，因此在运行make命令前，Vim可以自动保存文件。）一般而言，每当从命令模式下键入如下代码时：

```
:make arguments
```

Vim都会运行make并将*arguments*传递给它。

更大的好处是，当从Vim中编写程序时，编辑器能捕获编译器发出的所有消息。编辑器理解GCC输出内容的语法，知道何时发生编译器警告或错误。让我们再看一下实际应用，请看代码清单7-1。

代码清单7-1　*main.c*

```
#include <stdio.h>

int main(void)
{
   printf("There were %d arguments.\n", argc);

   if (argc .gt. 5) then
      print *, 'You seem argumentative today';
   end if

   return 0;
}
```

看来这个程序员既使用了Fortran又使用了C语言进行编码！假设当前在编辑*main.c*，要构建程序。从Vim中执行:make命令，并查看所有错误消息（图7-4）。

```
:!make  2>&1| tee /tmp/v243244/1
gcc -std=c99 -W -Wall    main.c   -o main
main.c: In function ‘main’:
main.c:12: error: ‘argc’ undeclared (first use in this function)
main.c:12: error: (Each undeclared identifier is reported only once
main.c:12: error: for each function it appears in.)
main.c:14: error: expected identifier before numeric constant
main.c:14: error: ‘then’ undeclared (first use in this function)
main.c:15: error: expected ‘;’ before ‘print’
main.c:15:18: warning: character constant too long for its type
main.c:16: error: ‘end’ undeclared (first use in this function)
main.c:16: error: expected ‘;’ before ‘if’
make: *** [main] Error 1
(3 of 12): error: ‘argc’ undeclared (first use in this function)
Press ENTER or type command to continue
```

图7-4　错误消息

现在，如果按下ENTER或空格键会返回到编辑程序，但是光标位于产生第一条警告或错误

的代码行上（在本例中，位于未声明argc的消息上），如图7-5所示。

```
#include <stdio.h>

int main(void)
{
   printf("There were %d arguments.\n", argc);

   if (argc .gt. 5) then
      print *, 'You seem argumentative today';
   end if

   return 0;
}
```

图7-5　光标位于第一个错误上

修复这个错误后，有两种方式前进到下一个错误。

- 可以重做程序，Vim会再次显示其余警告和错误，并将光标重新放到第一个错误上。这种方式适用于构建时间可以忽略不记的情况，尤其是用一个按键构建程序时，比如：

```
au BufNewFile,BufRead *.c map <F1> :make<CR>
```

- 也可以使用显示下一个错误或警告的命令:cnext。类似地，:cprevious显示上一个错误或警告，:cc显示当前错误或警告。这3个命令都很方便地将光标放在“活动”错误或警告的位置。

7.1.5　关于将文本编辑器作为IDE的最后一个考虑事项

虽然精通所选择的编辑器是不言自明的，以致人们往往会忽略这一点，但这确实是学习在特定环境下编程的第一步。毫不夸张地说，编辑器对于程序员，就好比乐器对于音乐家。即使是最富有创造性的作曲家，也需要知道弹奏乐器的基本知识，才能实现他们的想法，使其他人受惠。最大限度地学会使用编辑器可以更迅速地编写程序，更有效地领悟其他人的代码，并减少调试代码时需要执行的编译次数。

如果使用的是Vim，我们推荐你看看Steve Oualline所著的*Vi IMproved—Vim*（New Riders，2001）。这本书相当全面，写得也不错。（但是，它是为Vim 6.0编写的，Vim 7.0及其后版本的折叠等功能没有包括进去。)我们这里的目标只是初步了解一下Vim能够为程序员做的事。但Steve的著作是学习具体知识的极佳资源。

作者发现Vim有很多特别有用的功能。比如，本来我们还想讨论：

- 用K查询手册页面中的函数；
- 用gd和gD查找变量声明；
- 用[^D和]^D跳到宏定义；
- 用]d、]d、[D和]D显示宏定义；
- 划分窗口以同时查看.c和.h文件，以便检查原型；
- 其他种种。

但是本书是关于调试的，而不是关于Vim的。下面继续讨论其他软件工具。

7.2　充分利用编译器

如果说编辑器是对抗程序错误的第一个武器，那么编译器就是第二个武器。所有编译器都有能力扫描代码并查找常见错误，但是通常必须通过调用适当的选项来启用这种错误检查。

除了一些特殊情况，很多编译器警告选项（比如GCC的-Wtraditional开关）都显得有些大材小用。然而，在任何时候，不使用-Wall而使用GCC，最好连这样的想法都不要有。例如，新手C程序员最常犯的错误可以通过如下语句说明。

```
if (a = b)
   printf("Equality for all!\n");
```

这是有效的C代码，GCC会顺利地编译它。变量a被赋予值b，而且这个值用在条件语句中。然而，这肯定不是程序员的意思。使用GCC的-Wall切换，至少可以得到一条警报，指出这段代码可能有错误。

```
$ gcc try.c
$ gcc -Wall try.c
   try.c: In function `main':
   try.c:8: warning: suggest parentheses around assignment used as truth value
```

GCC建议在作为真正的值使用之前，将赋值语句a=b加上括号，与在赋值并执行如下比较时采用的方式一样：if ((fp = fopen("myfile", "w")) == NULL)。GCC实际上在问："你确认这里是要赋值a=b，而不是进行a==b的测试吗？"

应该总是使用编译器的错误检查选项。如果你在教授编程课，也应当要求学生使用这样的选项，以便逐渐灌输良好的习惯。GCC用户应当总是使用-Wall，即使在最小的"Hello, world！"程序中也是如此。我们发现，也要慎用-Wmissing-prototypes和-Wmissing-declarations。实际上，如果你有10分钟的空余时间，可以浏览一下GCC的手册页面，并读一下编译器警告部分，特别是要在Unix下编程时，很有必要。

7.3　C 语言中的错误报告

C语言中的错误报告是使用名为errno的老式机制完成的。虽然errno有点老态，而且有一些不足之处，但是一般都能完成工作。你可能会想，既然大部分C函数都有方便的返回值，可以表明调用是成功还是失败，为什么需要错误报告机制呢？答案是返回值可以提示你一个函数没有做你希望它做的事，但是它可能会也可能不会告诉你为什么。为了更具体地说明，来看如下代码片段。

```
FILE *fp
fp = fopen("myfile.dat", "r");
retval = fwrite(&data, sizeof(DataStruct), 1, fp);
```

假设你查看retval并发现它等于0。从手册页面上看到fwrite()应返回编写的项数（不是字

节或字符数），因此retval应为1。fwrite()会有多少种失败的方式？很多！首先，该文件可能已满，或者你可能没有在文件上写权限。然而，在本例中，代码中有导致fwrite()失败的程序错误。你能找出这个程序错误吗？[1]像errno这样的错误报告系统可以提供诊断信息，帮你找出在这样的情况下发生了什么事。（操作系统也可能宣告某些错误。）

使用errno

系统与库调用的失败通常会设置一个名为errno的全局定义整数变量。在大多数GNU/ Linux系统上，errno是在*/usr/include/errno.h*上声明的，因此，包括了这个头文件，就不必在源代码中声明extern int errno。

当一个系统或库调用失败时，它将errno设置为一个指示失败类型的值。检查errno的值，并采取适当的动作是你的职责。来看代码清单7-2。

代码清单7-2 *double-trouble.c*

```
#include <stdio.h>
#include <errno.h>
#include <math.h>

int main(void)
{
   double trouble = exp(1000.0);
   if (errno) {
      printf("trouble: %f (errno: %d)\n", trouble, errno);
      exit(-1);
   }

   return 0;
}
```

在我们的系统上，exp(1000.0)能够存储的值比double类型存储的值大，因此赋值导致浮点溢出。从输出中可以看出，errno的值34表示浮点溢出错误。

```
$ ./a.out
trouble: inf (errno: 34)
```

这很大程度上说明了errno的工作方式。根据惯例，当库函数或系统调用失败时，它将errno设置为一个描述调用失败原因的值。从上面的代码中看到，值34意味着exp(1000.0)的结果不能用double值表示，还有其他很多值表示下溢、权限问题、文件未找到等等错误情况。然而，在程序中开始使用errno之前，需要注意一些问题。

首先，使用errno的代码可能不是完全可移植的。例如，ISO C标准仅定义了少量错误代码，POSIX标准定义了很多错误代码。从errno手册页面中可以看到哪些错误代码是由哪些标准定义

7

① 我们用读模式打开了文件，然后试图向其中写信息。

的。而且，这些标准不是指定的像34这样的数值来表示错误代码，而是规定了符号错误代码，它们是一些以E为前缀的宏常量，在errno头文件中定义（或者在errno头文件所包括的文件中）。符号错误代码的值在不同平台上的唯一一致之处是它们都是非零值。因此，不能假定某个特定的值总是表示相同的错误情况。[①]应该总是使用符号名指代errno值。

除了ISO和POSIX errno值外，C库的特定实现（比如GNU的glibc）甚至可以定义更多errno值。在GNU/Linux上，libc的信息页面[②]的errno部分是该平台上所有可用errno值的规范来源：ISO、POSIX和glibc。下面是我们从GNU/Linux机器上提取出来的*/usr/include/asm/errno.h*中的部分错误代码定义。

```
#define EPIPE        32 /* Broken pipe */
#define EDOM         33 /* Math arg out of domain of func */
#define ERANGE       34 /* Math result not representable */
#define EDEADLK      35 /* Resource deadlock would occur */
#define ENAMETOOLONG 36 /* File name too long */
#define ENOLCK       37 /* No record locks available */
#define ENOSYS       38 /* Function not implemented */
```

接下来，关于errno用法还有一些要点要记住。errno可以通过任何库函数或系统调用设置，无论它是成功还是失败！因为成功的函数调用也能够设置errno，所以不能依赖errno告诉你是否发生了错误。只能依赖它指出为什么发生某个错误。因此，使用errno最安全的方式如下[③]。

(1) 执行对库或系统函数的调用。

(2) 使用函数的返回值判断是否发生了某个错误。

(3) 如果发生了某个错误，使用errno确定为什么发生这个错误。

用伪码表示如下。

```
retval = systemcall();

if (retval indicates an error) {              产生错误
   examine_errno();                           检查错误代号
   take_action();                             采取行动
}
```

然后要用到手册页面。假设你在编码，要在调用ptrace()后抛出一些错误检查。第二步表示，使用ptrace()的返回值来确定有没有发生某个错误。如果你像我们一样做，就不必记住ptrace()的返回值。可以怎么做呢？每个手册页面都有一个名为Return Value的部分。可以快速来到这个页面，键入man *function name*来搜索字符串return value。

虽然errno有一些缺陷，但也有一些优势。

GNU库在幕后需做大量工作，当进入函数时保存errno，然后如果函数调用成功，将它恢复

① 例如，有些系统中EWOULDBLOCK与EAGAIN是不同的，但是GNU/Linux对这两者不加区别。

② 除了lilbc信息页面外，也可以在系统头文件中查看errno值。这样不仅安全而且自然，实际上我们鼓励这样做！

③ 在代码清单7-2中对errno的使用不是好的做法。

为原始值。如果函数调用成功，glibc函数一般不会重写errno。然而，GNU还没有完全普及，因此可移植代码不应依赖于这一事实。

另外，虽然每次要查看特定错误代码时都过一遍文档太繁琐，但是有两个函数使得错误代码的解释更容易：perror()和strerror()。虽然它们做的事情相同，但是方式不同。perror()函数接受字符串参数，且没有返回值。

```
#include <stdio.h>
void perror(const char *s);
```

perror()的参数是用户提供的字符串。当调用perror()时，它输出这个字符串，后面跟着一个冒号和空格，然后是基于errno的值进行的错误类型描述。代码清单7-3是一个关于如何使用perror()的简单示例。

代码清单7-3 *perror-example.c*

```
int main(void)
{
   FILE *fp;

   fp = fopen("/foo/bar", "r");

   if (fp == NULL)
      perror("I found an error");

   return 0;
}
```

如果系统上没有文件*/foo/bar*，输出如下所示。

```
$ ./a.out
I found an error: No such file or directory
```

perror()的输出是一个标准错误。如果要将程序的错误输出重定向到一个文件中，需记住这一点。

另一个有助于将errno代码翻译成描述性消息的函数是strerror()。

```
#include <string.h>
char *strerror(int errnum);
```

这个函数将errno的值作为其参数，并返回一个描述错误的字符串。代码清单7-4是一个关于如何使用strerror()的示例。

代码清单7-4 *strerror-example.c*

```
int main(void)
```

```
{
  close(5);
  printf("%s\n", strerror(errno));
  return 0;
}
```

该程序的输出如下。

```
$ ./a.out
Bad file descriptor
```

7.4　更好地使用 strace 和 ltrace

了解库函数和系统调用之间的区别很重要。库函数是更高级别的，完全在用户空间里运行，并为程序员提供了到做实际工作（系统）的函数的更方便的接口。系统调用代表用户以内核模式工作，由操作系统本身的内核提供。库函数`printf()`看上去类似于一般输出函数，但是它实际上只是格式化你提供给字符串的数据，并用低级系统调用`write()`编写字符串数据，然后将数据发送到一个与终端的标准输出关联的文件中。

`strace`实用程序输出程序进行的各个系统调用及其参数和返回值。如果想查看`printf()`进行了哪些系统调用，很容易做到。编写一个“Hello, world！”程序，按如下代码运行它。

```
$ strace ./a.out
```

计算机很努力地工作，只是为了向屏幕上显示一些内容。

`strace`输出的每一行对应于一个系统调用。`strace`的大多数输出显示了对`mmap()`和`open()`的调用，采用类似*ld.so*和*libc*这样的文件名。它们与一些系统级操作有关，比如将磁盘文件映射到内存中及加载共享库等。一般来说不必关心所有系统操作。对于我们而言，只要关心靠近输出末尾的两行即可。

```
write(1, "hello world\n", 12hello world) = 12
_exit(0) = ?
```

这些代码行说明了`strace`输出的一般格式：

- 被调用的系统函数名；
- 圆括号括起来的系统调用参数；
- =号后面的系统调用[①]返回值。

就这些，但是包含的信息却相当丰富！你也可以发现错误。比如，在我们的系统上，得到如下代码行。

```
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or directory)
```

---

① 或许你会吃惊地发现`exit()`的返回值是问号。这里`strace`只是要表明`_exit`返回了一个`void`值。

```
open("/etc/ld.so.cache", O_RDONLY)   = 3
```

第一次调用open()是为了打开名为*/etc/ld.so.preload*的文件。对open()的调用得到的返回值（应当是非负的文件描述符）是-1，说明出现了某种错误。Strace热心地告诉我们导致open()失败的错误是ENOENT：文件*/etc/ld.so.preload*不存在。

strace告诉我们，对open()第二次调用的返回值为3。这是一个有效的文件描述符，因此显然打开文件*/etc/ld.so.cache*的调用成功了。相应地，strace输出的第二行上没有错误代码。

顺便说一下，不要担心像这样的错误。你看到的内容与动态库加载有关，本质上并不是真正的错误。文件*ld.so.preload*可用来重写系统的默认共享库。由于我无意纠缠这些事情，所以该文件根本不在我的系统上。当你逐渐获得strace的经验时，就会越来越懂得如何过滤掉这种“噪音”，将精力集中于真正感兴趣的输出部分。

strace有一些需要在某个时候使用的选项，因此我们这里简单地介绍一下。看一下“Hello, world!”程序的完整strace输出，可能注意到strace有一点冗长。将所有输出保存在一个文件中比试图在屏幕上查看要方便得多。当然，可以使用重定向stderr的方式，但是也可以用*-o logfile*开关，使strace将其所有输出都写到一个日志文件中。另外，strace一般将字符串截短为32个字符。这一特性有时会隐藏重要信息。为了强制strace将字符串截短为*N*个字符，可以使用-s *N*选项。最后，如果在具有子进程分支的程序上运行strace，可以用-o LOG –ff开关，将单个子进程的strace输出捕获到一个名为*LOG.xxx*的文件中，其中*xxx*是子进程ID。

还有一个名为ltrace的实用工具，它类似于strace，但它显示了库调用而不是系统调用。ltrace和strace有很多共同的选项，因此知道如何使用其中一个，就很容易学会使用另一个。

当你不具有源代码（甚至具有源文件）时，如果要向维护人员发送程序错误报告和诊断信息，strace和ltrace实用程序非常有用，使用这些工具有时比深度调试代码更快。

本书作者之一试图在他自己的系统上安装并运行一个文档不全的专有应用程序时，偶尔发现了这些工具的有用性。当启动时，似乎不需要做任何事情，应用程序就立即返回到shell。他想向公司发送一些比仅仅报告“你的程序即将退出”所含信息更丰富的内容。在该应用程序上运行strace产生了一个线索：

```
open(umovestr: Input/output error 0, O_RDONLY) = -1 EFAULT (Bad address)
```

运行ltrace产生了更多线索：

```
fopen(NULL, "r")                                = 0
```

从输出中可以发现，这是对fopen()的第一次调用。应用程序大概想打开某种配置文件，但是传递给fopen()的是NULL。应用程序有某种内部错误处理程序，退出但是不产生错误消息。这位作者因此向公司发送了详细的程序错误报告，而事实证明，问题在于应用程序配备了错误的全局配置文件，它指向了一个不存在的局部配置文件。第二天公司就发布了一个补丁。

从那以后，我们两位作者发现strace和ltrace对于跟踪程序错误和解决棘手且会引起很多麻烦的奇怪行为非常有用。

7.5 静态代码检查器：lint 与其衍生

市面上有很多扫描代码的免费或商业工具，不编译代码，仅仅警告错误、可能的错误和与严格C语言编码标准的差距，这样的工具称为静态代码检查器。C语言的规范静态代码检查器由S. C. Johnson在20世纪70年代末编写，称为lint。编写lint主要是为了检查函数调用，因为C语言的早期版本不支持原型。lint产生了很多衍生静态检查器。其中之一由美国弗吉尼亚大学计算机科学系的Dave Evans编写，名为lclint，在Linux等现代系统上广泛使用。2002年1月，Dave将lclint改名为splint，以强调对安全编程的重视（也因为*splint*比*lcint*更容易拼读）。

splint的目标是帮助编写最具防御性、安全和无错的程序。splint像它的前身一样，对于组成有效代码的内容非常挑剔。[①]作为练习，试找出代码清单7-5中可能引起警告的内容。

代码清单7-5　*scan.c*

```
int main(void)
{
   int i;

   scanf("%d", &i);

   return 0;
}
```

当通过splint运行代码时，发出一条警告，表示即将丢弃scanf()的返回值。[②]

```
$ splint test.c
Splint 3.0.1.6 --- 11 Feb 2002

test.c: (in function main)
test.c:8:2: Return value (type int) ignored: scanf("%d", &i)
   Result returned by function call is not used. If this is intended, can cast
   result to (void) to eliminate message. (Use -retvalint to inhibit warning)

   Finished checking --- 1 code warning
```

所有splint警告都有固定格式。splint警告的第一行指出发生警告的文件名和函数。下面一行提供了警告的行号和位置。接着是警告的描述，以及关于如何抑制这种警告的说明。可以看到，调用splint -retvalint test.c关闭所有关于丢弃整数函数返回值的警告。另外，如果不想关闭所有关于被丢弃的int返回值的报告，可以通过将scanf()的类型强制转换成void来抑制对scanf()的这一调用的警告。也就是用(void) scanf("%d", &i);替换scanf("%d", &i);。（还有另一种抑制这种警告的办法，即使用注解，感兴趣的读者可参见splint文档了解详情。）

---

① 很多程序员将splint没有报告警告看作一种极大的荣誉。当出现这种情况时，代码被声明为无lint的。

② scanf()的返回值是指定的输入项数。

7.5.1　如何使用splint

splint有很多开关，但是在使用时能感觉到哪些开关能满足需要。很多开关本质上是布尔值：它们打开或关闭某个功能。在这些类型的开关前面加上+就将它们打开，加上-就将它们关闭。例如，-retvalint关闭被丢弃的int返回值的报告，+retvalint打开这个报告（这是默认行为）。

有两种使用splint的办法。第一种是非正式的，主要用在针对其他人的代码或即将完成的代码运行splint时。

```
$ splint +weak *.c
```

如果没有+weak开关，splint通常因为太挑剔而用处不大。除了+weak外，还有3种级别的检查。参见splint手册了解关于它们的更多信息，这里简述如下：

- +weak，弱检查，通常用于无注解的C代码；
- +standard，默认模式；
- +checks，中度严格检查；
- +strict，高度严格检查。

splint的更正式的用途涉及对专门与splint一起使用的代码进行文档记录。这种splint特有的文档称为*注解*（annotation）。注解是一个大主题，本书没有足够的篇幅来介绍，但是可以参见splint文档了解详细信息。

7.5.2　本节最后注意事项

splint支持很多（但不是全部）C99库扩展。它也不支持部分C99语言变化。例如，splint不能处理复杂数据类型，也不知道如何在for循环的初始化器中定义int变量。

splint是在GNU GPL下发布的，其主页为*http://www.splint.org/*。

7.6　调试动态分配的内存

你可能知道，*动态分配的内存*（Dynamically Allocated Memory，DAM）是程序用malloc()和calloc()这样的函数从堆中请求的内存。[①]动态分配的内存通常用于二叉树和链表等数据结构，在面向对象编程中创建对象时，幕后也在动态分配内存。甚至标准C语言库在内部也使用DAM。你还可能记得，处理完后必须释放动态内存。[②]

众所周知，DAM问题相当难查找，分为以下几类。

- 没有释放动态分配的内存。
- 对malloc()的调用失败（这容易通过检查malloc()的返回值来检测）。
- 向DAM段之外的地址执行读和写操作。

① 在本节其余部分，我们仅提及malloc()，但实际上表示malloc()及其类似函数，比如calloc()和realloc()。

② 一个值得注意的例外是alloca()函数，它从当前栈帧请求动态内存，而不是从堆请求。当该函数返回时，自动释放帧中的内存。因此，你不必释放alloca()所分配的内存。

- 释放DAM段之后对DAM区域中的内存执行读写操作。
- 对动态内存的同一段调用两次`free()`。

这些错误可能不会导致程序以明显的方式崩溃。让我们进一步讨论这个问题。为了具体说明这些问题，来看代码清单7-6。

代码清单7-6　*memprobs.c*

```
int main( void )
{
   int *a = (int *) malloc( 3*sizeof(int) ); // malloc return not checked
   int *b = (int *) malloc( 3*sizeof(int) ); // malloc return not checked

   for (int i = -1; i <= 3; ++i)
      a[i] = i; // a bad write for i = -1 and 3

   free(a);
   printf("%d\n", a[1]); // a read from freed memory
   free(a); // a double free on pointer a

   return 0; // program ends without freeing *b.
}
```

第一个问题称为内存泄漏。例如，来看代码清单7-7。

代码清单7-7　内存泄漏的示例

```
int main( void )
{
        ... lots of previous code ...

        myFunction();
        ... lots of future code ...
}


void myFunction( void )
{
        char *name = (char *) malloc( 10*sizeof(char) );
}
```

当`myFunction()`执行时，为10个`char`值分配内存。引用这个内存的唯一办法是通过使用`malloc()`返回的地址（存储在指针变量`name`中）。如果由于某种原因丢失了这个地址，比如`myFunction()`退出时`name`超出了作用域，而且没有在别的地方保存其值的副本，那么就没有办法访问分配的内存，特别是无法释放它。

但这正是这段代码中发生的事情。动态分配的内存不会像`name`等为栈分配的存储器那样简单地消失或超出作用域，因此每次调用`myFunction()`时，都会消耗10个字符的内存，然后永远不释放。最终结果是变量堆空间变得越来越小。这就是人们将这种程序错误称为*内存泄漏*的原因。

内存泄漏减少了程序的可用内存数量。在大多数现代系统（比如GNU/Linux）上，当存在内存泄漏的应用程序终止时，操作系统会回收这种内存。在老的系统上（比如Microsoft DOS和Microsoft Windows 3.1），泄漏的内存会失去，直到重启系统才会回收。无论系统新旧，内存泄漏都会导致系统性能下降，因为增加了分页操作。随着时间的推移，它们会导致有内存泄漏的程序甚至整个系统崩溃。

动态分配内存遇到的第二个问题是对`malloc()`的调用可能失败。发生这种情况的方式很多。例如，计算中的程序错误可能导致请求大量的或为负值的DAM。或者，也许系统内存真的用完了。如果没有意识到发生了这个问题，而继续试图在你误以为有效的DAM上读写，则局面会雪上加霜。这是一种我们很快就会讨论的访问违反。然而，为了避免这种情况，应当总是检查`malloc()`有没有返回非`NULL`指针，如果没有返回这种指针，应尝试优雅地退出程序。

第三个和第四个问题称为*访问错误*。这两个问题本质上是同一件事情的不同版本：程序试图在不可用的内存地址上读或写。第三个问题涉及访问DAM段以上或以下的内存地址。第四个问题涉及访问过去曾经可用，但是在访问尝试之前被释放了的内存地址。

对DAM的同一个段调用两次`free()`俗称为*重复释放*（double free）。C库有种内部内存管理结构描述分配的每个DAM段边界。当对指向动态内存的同一个指针两次调用`free()`时，程序的内存管理结构被破坏，导致程序崩溃；在有些情况下，甚至会使怀有恶意的程序员利用这个程序错误产生缓冲区溢出。在某种意义上，这也是一种访问违反，但这是针对C库本身的，而不是针对程序的。

访问违反会导致发生这两件事情之一：一是程序崩溃，可能写一个核心文件[①]（通常在收到段错误信号后）；二是更糟糕的情况，程序能够继续执行，导致数据损坏。

在这两种结果中，前者绝对要好得多。事实上，有很多可用的工具会导致每当检测到DAM有任何问题时，程序发生段错误，并转储核心，这胜于冒第二种情况的风险！

你可能会问："我到底为什么要让程序发生段错误呢？"要知道，如果处理DAM的代码中有程序错误，那么让它崩溃是一件求之不得的事（Very Good Thing），否则将面临间歇性、令人费解且不能再现的坏行为。在内存损坏的影响被感觉到之前，可能很长时间都不被注意。通常情况下，这个问题在离程序错误相当远的程序部分中才表现出来，所以非常难以跟踪。更糟糕的是，内存损坏会导致安全性被破坏。已知的会导致缓冲区溢出和重复释放的应用程序，在有些情况下会被恶意黑客用来运行任意代码并攻击很多操作系统安全漏洞。

另一方面，当程序发生了段错误并转储了核心文件，可以对核心文件执行事后检查，了解导致该段错误的精确源文件及代码行号。这是更好的捕获程序错误的方式。

简言之，尽快捕获DAM问题极其重要。

① 这俗称转储核心。

7.6.1　检测DAM问题的策略

本节讨论Electric Fence，这是对分配的内存地址实施“栅栏”防护功能的库。访问这些栅栏外的内存通常会导致段错误和核心转储。本节还会讨论两个GNU工具`mtrace()`和`MALLOC_CHECK_`，它们向标准libc分配函数中添加钩子，以保持关于当前分配内存的记录。这样，libc就可以对要读、写或释放的内存执行检查。记住，在使用几个软件工具时要小心，每个工具都使用钩子进行与堆相关的函数调用，因为一个工具可以在已经安装的钩子上再安装一个钩子。[①]

7.6.2　Electric Fence

Electric Fence（即EFence），是Bruce Perens在1988年编写，在GNU GPL许可下发布的。那时他在Pixar工作。当EFence链接到代码中时，导致程序在发生下列情况之一时立即发生段错误并转储核心[②]。

- 在DAM边界之外执行读或写操作。
- 对已经释放的DAM执行读或写操作。
- 对没有指向`malloc()`分配的DAM的指针执行`free()`（包括重复释放的特殊情况）。

让我们来看如何使用Electric Fence跟踪`malloc()`问题。请看代码清单7-8。

代码清单7-8　*outOfBound.c*

```
int main(void)
{
   int *a = (int *) malloc( 2*sizeof(int) );

   for (int i=0; i<=2; ++i) {
      a[i] = i;
      printf("%d\n ", a[i]);
   }

   free(a);
   return 0;
}
```

虽然程序中包含原型的`malloc()`程序错误，但是它可能会没有抱怨地正常编译，甚至很可能没有问题地运行。[③]

---

① 实际上，可以安全地一起使用`mtrace()`和`MALLOC_CHECK_`，因为`mtrace()`小心地保留了它发现的任何现有钩子。

② 如果从GDB中运行链接了EFence的程序，而不是在命令行上调用它，那么程序会发生段错误，不转储核心。这正合你意，因为链接到EFence的可执行文件的核心文件会很大，而且你不需要核心文件，因为你总是位于GDB中，并正盯着发生段错误的源代码文件和行号处。

③ 这不意味着`malloc()`溢出不会对代码造成严重破坏！本例主要为了说明如何使用EFence。在实际问题中，编写超出数组边界的代码会导致一些严重问题！

```
$ gcc -g3 -Wall -std=c99 outOfBound.c -o outOfBound_without_efence -lefence
$ ./outOfBound_without_efence
0
1
2
```

我们能够在数组a[]的最后一个元素之外进行写操作。现在看上去一切正常，但是只能说明这个程序错误本身不可预知，以后难以确切地跟踪。

现在我们将outOfBound与EFence链接并运行它。默认情况下，EFence仅捕获对动态分配区域的最后一个元素之外的读或写操作。这意味着当试图写入a[2]时OutOfBound应当出现段错误。

```
$ gcc -g3 -Wall -std=c99 outOfBound.c -o outOfBound_with_efence -lefence
$ ./outOfBound_with_efence
  Electric Fence 2.1 Copyright (C) 1987-1998 Bruce Perens.
0
1
Segmentation fault (core dumped)
```

果然，EFence发现写操作超过了数组的最后一个元素。

虽然不小心访问数组第一个内存之前的内存（比如指定“元素”a[-1]）的情况不太常见，但是错误的索引计算可能会导致这样的情况。EFence提供了一个名为EF_PROTECT_BELOW的全局int变量。当将这个变量设置为1时，EFence仅捕获数组下溢，不检查数组上溢。

```
extern int EF_PROTECT_BELOW;


double myFunction( void )
{
   EF_PROTECT_BELOW = 1;  // Check from below

        int *a = (int *) malloc( 2*sizeof(int) );

        for (int i=-2; i<2; ++i) {
                a[i] = i;
                printf("%d\n", a[i]);
        }

   ...
}
```

由于EFence的这种工作方式，我们可以捕获试图访问超出动态分配的块的内存，或者试图访问分配的块之前的内存，但是不能同时捕获这两种类型的访问错误。

为了更彻底地捕获程序错误，应当使用EFence运行程序两次：一次在默认模式下检查动态内

存上溢，另一次将EF_PROTECT_BELOW设置为1来检查下溢。[①]

除了EF_PROTECT_BELOW外，可以设置EFence的另外几个全局整型变量，以控制其行为。

- EF_DISABLE_BANNER。将这个变量设置为1可以隐藏运行与EFence链接的程序时显示的横幅。我们不建议这样做，因为横幅可以提醒你EFence已链接到应用程序，不应将可执行文件用于生产版本，因为链接到EFence的可执行文件较大，运行较慢，并且会产生非常大的核心文件。
- EF_PROTECT_BELOW。正如我们讨论过的，EFence默认检查DAM上溢。将这个变量设置为1会使EFence检查内存下溢。
- EF_PROTECT_FREE。默认情况下，EFence不检查对已被释放DAM的访问。将这个变量设置成1会启用对已释放内存的保护。
- EF_FREE_WIPES。默认情况下，EFence不修改存储在已释放内存中的值。将这个变量设置成非零值导致EFence填充在释放前用0xbd动态分配的内存的段。这样使EFence更容易检测出对被释放的内存的不当访问。
- EF_ALLOW_MALLOC_0。默认情况下，EFence会捕获对malloc()的参数为0的任何调用（即对零字节内存的任何请求）。其基本原理是编写类似char *p = (char *) malloc(0);的代码可能是程序错误。然而，如果由于某些原因，你的意思真的是向malloc()传递0，那么将这个变量设置为非零值会使EFence忽略这样的调用。

作为练习，请编写一个访问已释放的DAM的程序，并使用EFence捕获这一错误。

每当修改这些全局变量之一时，都需要重新编译程序，这很不方便。幸好还有一种更容易的方式。也可以设置与EFence的全局变量同名的shell环境变量。EFence会检测这些环境变量，并采取适当的动作。

作为示范，我们将环境变量EF_DISABLE_BANNER设置为禁止显示EFence横幅页面。（正如以前所提到的，你不应这样做。）如果使用Bash，执行：

```
$ export EF_DISABLE_BANNER=1
```

C shell用户应执行：

```
% setenv EF_DISABLE_BANNER 1
```

然后重新运行代码清单7-8，并确认禁用了横幅。

另一个技巧是在调试会话期间从GDB中设置EFence变量。这样之所以可行，是因为 EFence变量是全局的。然而，这也意味着程序需要在执行中，却被暂停了。

7.6.3　用GNU C库工具调试DAM问题

如果使用的是GNU平台，比如GNU/Linux，那么可以用一些GNU C库特有的功能（类似于

① 欲知详情，请参阅EFence的手册页面的“Word-Alignment and Overrun Detection”和“Instructions for Debugging Your Program” 部分。

EFence）来捕获动态内存问题并修复。我们下面简单介绍一下。

1. **MALLOC_CHECK_Environment变量**

GNU C库提供了一个名为MALLOC_CHECK_的环境变量，像EFence一样，可用来捕获DAM访问违反，但是对它的使用不需要重新编译程序。这些设置及其效果如下所示。

- 0——关闭所有DAM检查（如果没有定义变量，也是这种情况）。
- 1——当检测到堆损坏时，显示关于stderr的诊断消息。
- 2——当检测到堆损坏时，立即退出程序并转储内存。
- 3——1和2的综合效果。[①]

由于MALLOC_CHECK_是环境变量，因此使用它查找与堆相关的问题很简单，只要键入如下代码即可：

```
$ export MALLOC_CHECK_=3
```

虽然MALLOC_CHECK_比EFence用起来更方便，但是它有几个严重的缺陷。第一，MALLOC_CHECK_仅在下次执行与堆相关的函数（比如malloc()、realloc()或free()）时出现非法内存访问的情况下才报告动态内存问题。这意味着你不仅不知道有问题代码的源文件和行号，而且经常甚至不知道是哪个指针引起了问题。为了说明这一点，来看代码清单7-9。

代码清单7-9 *malloc-check-0.c*

```
1  int main(void)
2  {
3     int *p = (int *) malloc(sizeof(int));
4     int *q = (int *) malloc(sizeof(int));
5
6     for (int i=0; i<400; ++i)
7        p[i] = i;
8
9     q[0] = 0;
10
11    free(q);
12    free(p);
13    return 0;
14 }
```

在第11行出现的程序异常中断实际上问题发生在第7行。分析一下核心文件可以确认问题位于q上，而不是p上。

```
$ MALLOC_CHECK_=3 ./malloc-check-0
malloc: using debugging hooks
```

① 这在作者的系统上没有文档说明。感谢Gianluca Insolvibile阅读glibc源代码并发现这个选项！

```
free(): invalid pointer 0x8049680!
Aborted (core dumped)
$ gdb malloc-check-0 core
Core was generated by `./malloc-check-0'.
Program terminated with signal 6, Aborted.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x40046a51 in kill () from /lib/libc.so.6
(gdb) bt
#0  0x40046a51 in kill () from /lib/libc.so.6
#1  0x40046872 in raise () from /lib/libc.so.6
#2  0x40047986 in abort () from /lib/libc.so.6
#3  0x400881d2 in _IO_file_xsputn () from /lib/libc.so.6
#4  0x40089278 in free () from /lib/libc.so.6
#5  0x080484bc in main () at malloc-check-0.c:13
```

在调试这个只有14行的程序时，你还能够应付这一缺陷，但是当使用真正的应用程序时，这可能是一个严重问题。然而，知道DAM问题存在，这本身是一个有用信息。

其次，这暗示了如果访问错误发生后没有调用与堆相关的函数，那么`MALLOC_CHECK_`根本不会报告错误。

再次，`MALLOC_CHECK_`错误消息似乎含义不是那么明显。虽然前面的代码清单有一个数组上溢错误，但是这个错误消息仅仅指出是“无效指针”。这从技术上来说是正确的，然而用处不大。

最后，对于setuid和setgid程序是禁用`MALLOC_CHECK_`的，因为怀有恶意的人可以利用这种功能组合进行安全攻击。可以通过创建文件*/etc/suid-debug*来重新启用。这个文件的内容不重要，重要的是存在这个文件。

总而言之，`MALLOC_CHECK_`是一种方便的工具，在代码开发期间用来捕获与堆相关的编程故障。然而，如果怀疑有DAM问题，或者要仔细扫描代码查找可能存在的DAM问题，就应当使用另一个实用工具。

2. 使用mcheck()工具

除`MALLOC_CHECK_`外还可以用`mcheck()`工具。我们发现这个方法比`MALLOC_CHECK_`更令人满意。`mcheck()`的原型为：

```
#include <mcheck.h>
int mcheck (void (*ABORTHANDLER) (enum mcheck_status STATUS))
```

在调用任何与堆相关的函数前必须调用`mcheck()`，否则对`mcheck()`的调用会失败。因此，应当在程序中非常早的时候调用这个函数。一旦成功调用`mcheck()`就会返回0，如果调用得太迟则会返回−1。

检测到DAM中的不一致时会调用用户提供的函数，参数`*ABORTHANDLER`是指向这个函数的指

针。如果将NULL传递给mcheck()，则使用默认处理程序。与MALLOC_CHECK_一样，这个默认处理程序也向stdout输出一条错误消息，并调用abort()来产生核心文件。与MALLOC_CHECK_不同的是，这条错误消息是有用的。例如，在代码清单7-10中，超出动态分配的段的末尾。

代码清单7-10 *mcheckTest.c*

```
int main(void)
{
   mcheck(NULL);
   int *p = (int *) malloc(sizeof(int));
   p[1] = 0;
   free(p);
   return 0;
}
```

它产生如下所示的错误消息。

```
$ gcc -g3 -Wall -std=c99 mcheckTest.c -o mcheckTest -lmcheck
$ ./mcheckTest
memory clobbered past end of allocated block
Aborted (core dumped)
```

其他类型的问题有类似的描述性错误消息。

3. 使用mtrace()捕获内存泄漏和重复释放

mtrace()工具是GNU C库的一部分，用来捕获C和C++程序中的内存泄漏和重复释放。mtrace()的使用涉及5个步骤。

(1) 将环境变量MALLOC_TRACE设置成有效的文件名。这是mtrace()在其中放置消息的文件名。如果没有将该变量设置为有效文件名，或者没有设置该文件的写权限，那么mtrace()将什么也不做。

(2) 包括*mcheck.h*头文件。

(3) 在程序最上方调用mtrace()。其原型为：

```
#include <mcheck.h>
void mtrace(void);
```

(4) 运行程序。如果检测到任何问题，会用一种非人类可读的形式将它们记到MALLOC_TRACE所指向的文件中。另外，为了安全起见，mtrace()不会对setuid或setgid可执行文件做任何事情。

(5) mtrace()配备了一个称为mtrace的Perl脚本，用来分析日志文件，并将其内容显示为人类可读的标准输出形式。

注意，这里还调用muntrace()来停止内存跟踪，但是glibc信息页面建议不要使用这个调用。也可能对程序使用了DAM的C库会得到通知，只有当main()已返回或进行了对exit()的调用后程序才会结束。在发生这件事之前，C库为程序使用的内存不会被释放。在该内存释放之前对

7

muntrace()的调用可能导致判断错误。

看一个简单示例。代码清单7-11说明mtrace()捕获到了这两个问题。在下面的代码中，我们一直没有释放第6行上p指向的内存，在第10行上我们在指针q上调用free()，尽管它没有指向动态分配的内存。

代码清单7-11 *mtrace1.c*

```
1  int main(void)
2  {
3     int *p, *q;
4
5     mtrace();
6     p = (int *) malloc(sizeof(int));
7     printf("p points to %p\n", p);
8     printf("q points to %p\n", q);
9
10    free(q);
11    return 0;
12 }
```

设置MALLOC_TRACE变量后编译该程序并运行它。

```
$ gcc -g3 -Wall -Wextra -std=c99 -o mtrace1 mtrace1.c
$ MALLOC_TRACE="./mtrace.log" ./mtrace1
p points to 0x8049a58
q points to 0x804968c
```

如果看一下*mtrace.log*的内容，会发现它根本没有意义。然而，运行Perl脚本mtrace()产生了无法理解的输出。

```
$ cat mtrace.log
= Start
@ ./mtrace1:(mtrace+0x120)[0x80484d4] + 0x8049a58 0x4
@ ./mtrace1:(mtrace+0x157)[0x804850b] - 0x804968c
p@satan$ mtrace mtrace.log
- 0x0804968c Free 3 was never alloc'd 0x804850b

Memory not freed:
-----------------
   Address     Size     Caller
0x08049a58      0x4  at 0x80484d4
```

然而，这只是略有帮助，因为虽然mtrace()发现了问题，但是它将它们报告成了指针地址。所幸，mtrace()能够做得更好。mtrace()脚本也将该可执行文件的文件名作为可选的参数。使用

这个选项，得到了行号以及相关的问题。

```
- 0x0804968c Free 3 was never alloc'd
/home/p/codeTests/mtrace1.c:15

Memory not freed:
-----------------
   Address     Size     Caller
0x08049a58      0x4  at /home/p/codeTests/mtrace1.c:11
```

现在，这正是我们想看到的信息！

像MALLOC_CHECK_和mcheck()实用程序一样，mtrace()不能防止程序崩溃。它仅仅是检查问题。如果程序崩溃，mtrace()肯定会有一部分输出丢失或错乱，这可能产生令人迷惑的错误报告。处理这种情况的最佳办法是捕获并处理段错误，以便让mtrace()优雅地终止。代码清单7-12说明了如何做这件事。

代码清单7-12 *mtrace2.c*

```
void sigsegv_handler(int signum);

int main(void)
{
   int *p;

   signal(SIGSEGV, sigsegv_handler);
   mtrace();
   p = (int *) malloc(sizeof(int));


   raise(SIGSEGV);
   return 0;
}

void sigsegv_handler(int signum)
{
   printf("Caught sigsegv: signal %d. Shutting down gracefully.\n", signum);
   muntrace();
   abort();
}
```

第 8 章

对其他语言使用GDB/DDD/Eclipse



人们一般都知道GDB和DDD是C/C++程序的调试器，但是它们也可以用于其他语言的开发。Eclipse最初是为Java开发设计的，但是它为其他很多语言提供了插件。本章将介绍如何使用这种多语言能力。

GDB/DDD/Eclipse不一定是哪种特定语言的“最佳”调试器。每种特定的语言都有很多优秀调试工具可用。然而我们要指出的是，无论使用哪种语言编写程序，不管是C、C++、Java、Python、Perl还是其他可使用这些工具的语言/调试器，如果能够使用相同的调试界面，那将是相当棒的。DDD就适用于所有这些语言。

以Pyton为例。Python解释器本身包括一个简单的基于文本的调试器。同样，虽然也存在很多优秀的Python特有的GUI调试器和IDE，但是另一种选择是使用DDD作为Python内置调试器的界面。这样可以既获得GUI的便利，又仍然使用进行C/C++编码时熟悉的界面（DDD）。

这些工具的多语言功能是如何实现的呢？

- 虽然GDB最初是作为C/C++的调试器创建的，但是后来使用GNU的人也提供了一款Java编译器：GCJ。
- DDD本身不是调试器，而是GUI可以通过它来向底层调试器发布命令。对于C/C++，该底层调试器通常是GDB。然而，DDD经常可用来作为其他语言特有的调试器的前端。
- Eclipse也只是前端。各种语言的插件使它可以开发与调试用那些语言编写的代码。

本章将概述在Java、Perl、Python和汇编语言中用这些工具进行调试。应当指出的是，在各种情况下，都有一些别的功能是这里没有涉及的，我们建议你探索你在使用的语言的详细信息。

8.1 Java

以一个操作链表的应用程序为例。这里，类`Node`的对象表示数字链表中的节点，以键值升序排列。这个列表本身是类`LinkedList`的对象。测试程序*TestLL.java*从命令行读入数字，建立一个由这些数字组成的排序列表，然后输出这个排序列表。下面是源代码。

TestLL.java

```
 1  // usage:  [java] TestLL list_of_test_integers
 2
 3  // simple example program; reads integers from the command line,
 4  // storing them in a linear linked list, maintaining ascending order,
 5  // and then prints out the final list to the screen
 6
 7  public class TestLL
 8  {
 9     public static void main(String[] Args) {
10        int NumElements = Args.length;
11        LinkedList LL = new LinkedList();
12        for (int I = 1; I <= NumElements; I++)  {
13           int Num;
14           // do C's "atoi()", using parseInt()
15           Num = Integer.parseInt(Args[I-1]);
16           Node NN = new Node(Num);
17           LL.Insert(NN);
18        }
19        System.out.println("final sorted list:");
20        LL.PrintList();
21     }
22  }
```

LinkedList.java

```
 1  // LinkedList.java, implementing an ordered linked list of integers
 2
 3  public class LinkedList
 4  {
 5     public static Node Head = null;
 6
 7     public LinkedList()  {
 8        Head = null;
 9     }
10
11     // inserts a node N into this list
12     public void Insert(Node N)  {
13        if (Head == null)  {
14           Head = N;
15           return;
16        }
17        if (N.Value < Head.Value)  {
18           N.Next = Head;
```

```
19            Head = N;
20            return;
21         }
22         else if (Head.Next == null)  {
23            Head.Next = N;
24            return;
25         }
26         for (Node D = Head; D.Next != null; D = D.Next)  {
27            if (N.Value < D.Next.Value)  {
28               N.Next = D.Next;
29               D.Next = N;
30               return;
31            }
32         }
33      }
34
35      public static void PrintList()  {
36         if (Head == null) return;
37         for (Node D = Head; D != null; D = D.Next)
38            System.out.println(D.Value);
39      }
40   }
```

Node.java

```
1    // Node.java, class for a node in an ordered linked list of integers
2
3    public class Node
4    {
5       int Value;
6       Node Next;  // "pointer" to next item in list
7
8       // constructor
9       public Node(int V)  {
10         Value = V;
11         Next = null;
12      }
13   }
```

该代码中有一个程序错误，让我们试着找出这个错误。

8.1.1 直接使用GDB调试Java

人们一般将Java看作一种解释语言，但是使用GNU的GCJ编译器，可以将Java源代码编译为本地机器代码。这样可以使Java应用程序运行得快得多，这也意味着可以使用GDB进行调试。（确

保使用GDB 5.1或者更高的版本。）GDB本身或者通过DDD，比JBD（Java Development Kit配套的调试器）功能更强大。例如，JDB不允许设置条件断点，而前面介绍过，这是一种基本GDB调试技术。因此不仅可以少学一个调试器，而且获得更好的功能性。

首先将应用程序编译成本地机器码。

```
$ gcj -c -g Node.java
$ gcj -c -g LinkedList.java
$ gcj -g --main=TestLL TestLL.java Node.o LinkedList.o -o TestLL
```

这几行代码类似于普通GCC命令，除了`-main=TestLL`选项外，它指定函数`main()`将作为程序执行入口点的类。（我们一次编译两个源文件。我们发现，为了确保GDB正确地跟踪源文件，这是必需的。）针对测试输入运行程序后得到如下结果。

```
$ TestLL 8 5 12
final sorted list:

5
8
```

不知什么原因，输入的12消失了。让我们来看如何使用GDB来查找这个程序错误。像惯常一样启动GDB，首先告诉它，当Java的无用单元收集操作产生Unix信号时，不要停止或向屏幕输出公告。这种动作是一种干扰，可能干扰使用GDB单步调试的能力。

```
(gdb) handle SIGPWR nostop noprint
Signal        Stop      Print   Pass to program Description
SIGPWR        No        No      Yes             Power fail/restart
(gdb) handle SIGXCPU nostop noprint
Signal        Stop      Print   Pass to program Description
SIGXCPU       No        No      Yes             CPU time limit exceeded
```

现在，既然第一个明显的程序错误是输入中的数字12，那么让我们在`Insert()`方法开头设置一个断点，这个节点键值上的条件等于12。

```
(gdb) b LinkedList.java:13 if N.Value == 12
Breakpoint 1 at 0x8048bb4: file LinkedList.java, line 13.
```

另外，也可以尝试如下命令：

```
(gdb) b LinkedList.java:Insert if N.Value == 12
```

然而，虽然这在以后会起作用，但这时还没有加载`LinkedList`类。

现在在GDB中运行程序：

```
(gdb) r 8 5 12
Starting program: /debug/TestLL 8 5 12
[Thread debugging using libthread_db enabled]
```

```
[New Thread -1208596160 (LWP 12846)]
[New Thread -1210696800 (LWP 12876)]
[Switching to Thread -1208596160 (LWP 12846)]

Breakpoint 1, LinkedList.Insert(Node) (this=@47da8, N=@11a610)
    at LinkedList.java:13
13              if (Head == null)  {
Current language:  auto; currently java
```

根据确认原则，让我们确认要插入的值为12。

```
(gdb) p N.Value
$1 = 12
```

现在单步调试该代码。

```
(gdb) n
17              if (N.Value < Head.Value)  {
(gdb) n
22              else if (Head.Next == null)  {
(gdb) n
26              for (Node D = Head; D.Next != null; D = D.Next)  {
(gdb) n
27                 if (N.Value < D.Next.Value)  {
(gdb) p D.Next.Value
$2 = 8
(gdb) n
26              for (Node D = Head; D.Next != null; D = D.Next)  {
(gdb) n
12           public void Insert(Node N)  {
(gdb) n
33           }
(gdb) n
TestLL.main(java.lang.String[]) (Args=@ab480) at TestLL.java:12
12              for (int I = 1; I <= NumElements; I++)  {
```

这不好。我们遍历了所有`Insert()`，而没有插入12。

再仔细一点查看，你将发现在*LinkedList.java*中第26行开始的循环，我们将要插入的值12与列表中目前的两个值5和8相比较，发现在这两种情况下新值较大。这实际上就是错误所在。我们没有处理在其中要插入的值大于列表中所有已存在的值的情况。所以需要在第31行后面添加代码来处理这种情况。

```
else if (D.Next.Next == null)  {
   D.Next.Next = N;
   return;
}
```

纠正后，可以发现程序工作正常了。

8.1.2 使用DDD与GDB调试Java

如果使用DDD作为GDB的界面，那么这些步骤会容易得多，也更令人愉快（同样假定源代码是用GCJ编译的）。像惯常一样启动DDD。

```
$ ddd TestLL
```

（忽略关于临时文件的错误消息。）

源代码没有立即出现在DDD的源文本窗口中，因此可以首先使用DDD的控制台。

```
(gdb) handle SIGPWR nostop noprint
(gdb) handle SIGXCPU nostop noprint
(gdb) b LinkedList.java:13
Breakpoint 1 at 0x8048b80: file LinkedList.java, line 13.
(gdb) cond 1 N.Value == 12
(gdb) r 8 5 12
```

这时源代码出现了，当前正位于断点上。然后可以像惯常一样继续进行DDD操作。

8.1.3 使用DDD作为JDB的GUI

DDD可以直接与Java Development Kit的JDB调试器结合起来使用。如下命令：

```
$ ddd -jdb TestLL.java
```

会启动DDD，然后DDD调用JDB。

然而，我们发现它变得很麻烦，因此不建议采用DDD的这种用法。

8.1.4 用Eclipse调试Java

如果最初下载并安装的是Eclipse的适用C/C++开发版本，则需要获取JDT（Java Development Tools）插件。

基本操作与前面描述C/C++时一样，但是要注意如下几点。

- 当创建项目时，确保选择Java Project。另外，建议选中Use Project Folder As Root for Sources and Classes复选框。
- 使用导航器的Package Explorer视图。
- 一旦保存（或导入），源文件就会被编译成*.class*。
- 当创建运行对话框时，右击Java Application并选择New。在标为Main Class的空格中，填充在其中定义`main()`的类。
- 当创建调试对话框时，选中Stop in Main复选框。
- 在调试运行中，Eclipse会在`main()`之前停止。只要单击Resume按钮就可以继续。

图8-1显示了Eclipse中的一个典型Java调试场景。注意，与C/C++一样，按下`N`旁边的下三角

形，在Variables视图中会显示节点N的值。

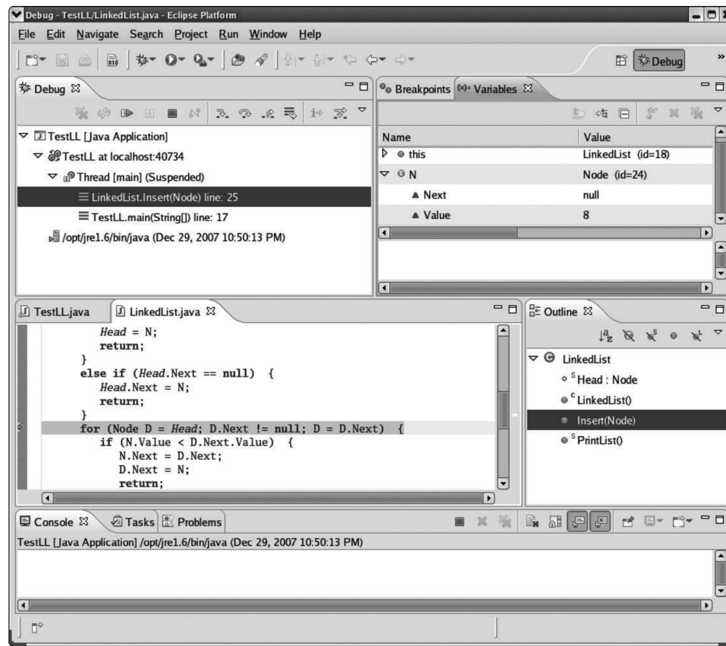


图8-1 Eclipse中的Java调试

8.2 Perl

我们将使用下面的示例*textcount.pl*，它计算文本文件上的统计信息。

```
1  #! /usr/bin/perl
2
3  # reads the text file given on the command line, and counts words,
4  # lines, and paragraphs
5
6  open(INFILE,@ARGV[0]);
7
8  $line_count = 0;
9  $word_count = 0;
10 $par_count = 0;
11
12 $now_in_par = 0;  # not inside a paragraph right now
13
14 while ($line = <INFILE>) {
15    $line_count++;
16    if ($line ne "\n")  {
17       if ($now_in_par == 0)  {
```

```
18            $par_count++;
19            $now_in_par = 1;
20         }
21         @words_on_this_line = split(" ",$line);
22         $word_count += scalar(@words_on_this_line);
23      }
24      else  {
25         $now_in_par = 0;
26      }
27   }
28
29   print "$word_count $line_count $par_count\n";
```

该程序统计在命令行上指定文件中的单词、行和段落数量。作为测试用例，我们使用如下所示的文件*test.txt*（你可能觉得似曾相识，原来它是本书第1章中的内容）。

```

In this chapter we set out some basic principles of debugging, both
general and also with regard to the GDB and DDD debuggers.  At least one
of our ``rules'' will be formal in nature, The Fundamental Principle of
Debugging.


Beginners should of course read this chapter carefully, since the
material here will be used throughout the remainder of the book.

Professionals may be tempted to skip the chapter.  We suggest, though,
that they at least skim through it.  Many professionals will find at
least some new material, and in any case it is important that all
readers begin with a common background.
```

在最上面有一行空白，*Debugging*后面有两行，*Professionals*前面有一行。在这个文件上运行我们的Perl代码后得到的输出应如下所示。

```
$ perl textcount.pl test.txt
102 14 3
```

现在，假设我们忘记了`else`子句。

```
else {
   $now_in_par = 0;
}
\end{Code}

The output would then be

\begin{Code}
$ perl textcount.pl test.txt
```

```
102 14 1
```

单词数量和行数是正确的，但段落数是错误的。

8.2.1　通过DDD调试Perl

Perl有它自己的内置调试器，这个调试器通过命令行上的-d选项调用。

```
$ perl -d myprog.pl
```

这个内置调试器的一个缺陷是它没有GUI，但是通过DDD运行调试器可以弥补这一缺陷。让我们看看可以如何使用DDD查找程序错误。键入如下代码来调用DDD：

```
$ ddd textcount.pl
```

DDD自动注意到这是Perl脚本，调用Perl调试器，并在第一行可执行代码上设置绿色的箭头表示当前的位置。

现在通过单击Program→Run来指定命令行参数test.txt，并填充弹出窗口的Run with Arguments部分，如图8-2所示。（在DDD控制台上可能会发现“…do not know how to create a new TTY…”这样的消息，忽略它。）另外，可以通过在DDD控制台中简单地键入如下Perl调试命令来“手工”设置参数：

```
@ARGV[0] = "test.txt"
```

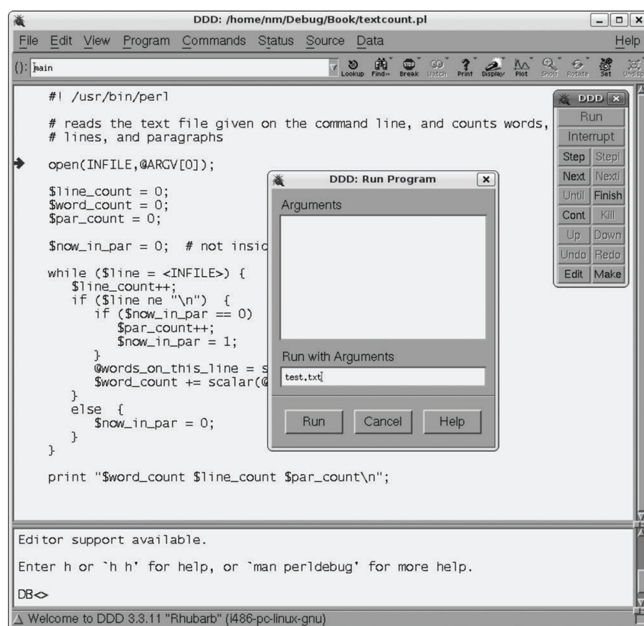


图8-2　设置命令行参数

由于错误在于段落数量中，因此让我们看看当程序达到第一段末尾时发生了什么事。这会在如下条件为真之后发生：

```
$words_on_this_line[0] eq "Debugging."
```

让我们在`while`循环开头附近放置一个断点。我们这样做的方式与前面见过的用于C/C++程序的方式完全相同，通过右击这一行，选择Set Breakpoint。

还应在这个断点上加上上面的条件。同样，单击Source→Breakpoints，确保突出显示Breakpoints and Watchpoints弹出窗口中的给定断点，然后单击Props图标。之后填充所需的条件，见图8-3。

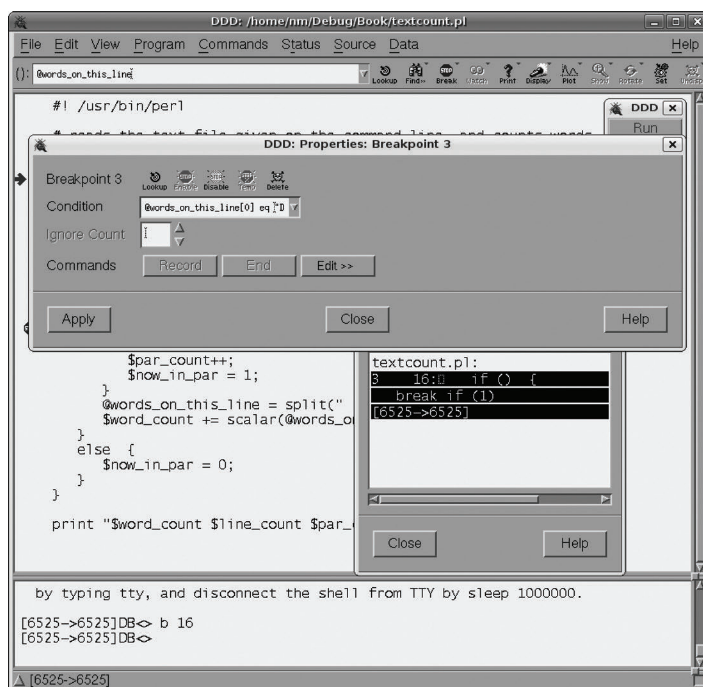


图8-3　在断点上设置条件

选择Program→Run。（我们不选择Run Again，因为这似乎把人带到了Perl内部。）我们将鼠标移到使它指向变量`$words_on_this_line`的实例，DDD会照常弹出黄色框显示该变量的值。因此我们确认断点条件满足，见图8-4。

当按下Next几次来跳过文本文件中的空白行后，你将注意到我们也跳过了下面这行。

```
$par_count++;
```

我们原来预期使用上面这行代码递增段落数。回头来看，这是由于变量`$now_in_par`为0引起的，通过观察，很快可以明白如何修复这一程序错误。

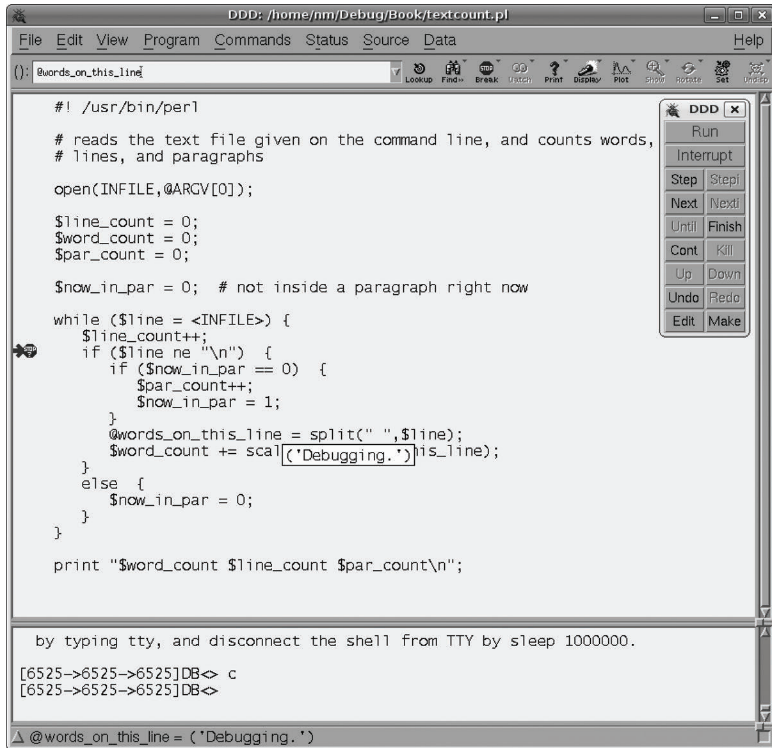


图8-4　到达断点时的情况

在DDD中，也可以通过选择Source→Breakpoints，突出显示断点，然后单击所需选项来禁用、启用或删除断点。

如果修改了源文件，则必须通知DDD通过选择File→Restart来更新其自身。

8.2.2　在Eclipse中调试Perl

为了在Eclipse中开发Perl代码，我们需要PadWalker Perl软件包（可以从CPAN下载），以及Perl的EPIC Eclipse插件。

同样，基本操作与以前对C/C++的描述相同，但是注意以下几点。

- 创建项目时，选择Perl Project。
- 使用Navigator作为导航器视图。
- 没有构建阶段，因为Perl不产生字节码。
- 在Debug透视图中，只能在Variables视图中访问变量的值，而不能通过在源代码视图中的鼠标动作来访问，而且只能访问局部变量。

需要在源代码中各个变量的第一个实例中使用Perl的my关键字来查看全局变量（这时会用到PadWalker包）。

图8-5显示了一个典型的Perl调试屏幕。注意我们向全局变量中添加了my关键字。

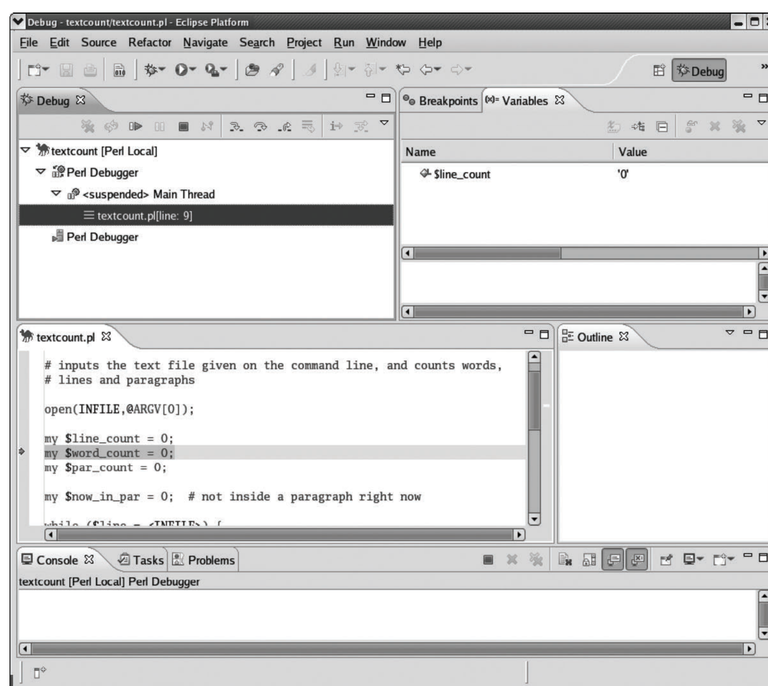


图8-5 Perl调试屏幕

8.3 Python

与上面的Perl示例一样，让我们以*tf.py*为例，统计文本文件中的单词数量、行数和段落数量。

```
1  class textfile:
2     ntfiles = 0  # count of number of textfile objects
3     def __init__(self,fname):
4        textfile.ntfiles += 1
5        self.name = fname  # name
6        self.fh = open(fname)
7        self.nlines = 0  # number of lines
8        self.nwords = 0  # number of words
9        self.npars = 0  # number of words
10       self.lines = self.fh.readlines()
11       self.wordlineparcount()
12    def wordlineparcount(self):
13       "finds the number of lines and words in the file"
14       self.nlines = len(self.lines)
15       inparagraph = 0
16       for l in self.lines:
17          w = l.split()
```

```
18            self.nwords += len(w)
19            if l == '\n':
20               if inparagraph:
21                  inparagraph = 0
22            elif not inparagraph:
23               self.npars += 1
24               inparagraph = 1
25
26      def grep(self,target):
27         "prints out all lines in the file containing target"
28         for l in self.lines:
29            if l.find(target) >= 0:
30               print l
31            print i
32
33   def main():
34      t = textfile('test.txt')
35      print t.nwords, t.nlines, t.npars
36
37   if __name__ == '__main__':
38      main()
```

一定要在源代码程序末尾包括如下所示的两行代码。

```
if __name__ == '__main__':
   main()
```

说明　如果你是经验丰富的Python程序员，很可能了解这个模式。如果你不熟悉Python，那么我解释一下，这几行代码是需要测试Python程序是本身被调用来还是被作为模块导入到了其他某个程序中。这是通过调试器运行程序时产生的一个问题。

8.3.1　在DDD中调试Python

Python的基本调试器是PDB（*pdb.py*），这是一个基于文本的工具。它的实用性通过使用DDD作为GUI前端而得到大大增强。

然而在开始前要先关注几个问题。为了使DDD正确地访问PDB，Richard Wolff编写了PYDB（*pydb.py*），这是对PDB略微修改后的版本。然后用`-pydb`选项运行DDD。但是随着Python的演变，原来的PYDB不再能正确地工作。

一个好的解决方案是Rocky Bernstein开发的。在编写本书时（2007年夏天），据说他的修改后的PYDB有了很大扩展，将包括到DDD的下一个版本中。另外，可以使用Richard Wolff提供的补丁。需要的文件如下：

- *http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydb.py*

- *http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbcmd.py*
- *http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbsupt.py*

将这些文件放在某个目录中，假设是*/usr/bin*，为它们赋予执行权限，并创建一个由下面这行代码组成的文件，命名为*pydb*：

```
/usr/bin/pydb.py
```

一定也要赋予*pydb*执行权限。完成这件事后就可以将DDD用于Python程序，然后激活DDD：

```
$ ddd --pydb
```

说明 如果弹出一个错误窗口，表明“PYDB不能启动”，可能是因为路径问题。例如，可能有另一个名为pydb的文件。只要确保在搜索路径中DDD的该文件是第一个遇到的文件。

然后打开源文件*tf.py*，可以通过选择File→Open Source并双击文件名，也可以通过在控制台中键入如下代码：

```
file tf.py
```

（Pdb）提示符最初可能不会显示，但是无论如何都要键入你的命令。

源代码会出现在源文本窗口中，然后可以像惯常一样设置断点。假设在如下这行代码上设置了一个断点：

```
w = l.split()
```

为了运行程序，单击Program→Run，在Run弹出窗口中设置任何命令行参数，然后单击Run。在单击Run按钮后，需要单击Continue两次。这是因为底层PDB/PYDB调试器的工作机理。（如果忘记了做这件事，PDB/PYDB会在DDD控制台中提醒你。）

要对源代码进行修改，则在控制台中执行`file`命令，或者选择Source→Reload Source。上次运行的断点会保留。

8.3.2 在Eclipse中调试Python

再次，Python的过程类似于其他语言。这当然说明了让同一个工具适用于多种语言的价值。一旦学会了如何在一种语言中使用某种工具，就很容易为另一种语言使用这种工具。关于Python特有的要点需记住如下内容。

- 需要安装Pydev插件。安装后，选择Window→Preferences→Pydev，并将Python解释器的位置（比如*/usr/bin/python*）通知Eclipse。
- 使用Pydev Package Explorer作为导航器透视图。
- 因为没有创建永久字节码文件，所以没有构建过程。
- 在建立运行/调试对话框时，注意如下内容：用希望从中开始执行的源文件名填充Main Module。在Arguments选项卡中，在Base Directory中填充具有输入文件的目录（或者这些

目录树的根目录），并将命令行参数放在Program Arguments中。

- 在Debug透视图中，变量的值只能通过Variables视图访问，而且仅限于访问局部变量。

Eclipse优于DDD的一个主要优点是，DDD使用的底层调试引擎PDB不适用于多线程程序，而Eclipse适用于这种程序。

8.4　调试 SWIG 代码

SWIG（Simplified Wrapper and Interface Generator）是一种流行的开源工具，用来将Java、Perl、Python和若干其他解释语言与C/C++接合。大部分Linux发行版都包括SWIG，也可以从网上下载。它允许使用解释语言编写应用程序的大部分代码，并与程序员用C/C++编写的特定部分结合，从而增强性能。

问题在于如何在这样的代码上运行GDB/DDD。下面我们将提供一个使用Python和C的小示例。C代码将管理一个先进先出（FIFO）队列。

```
1  // fifo.c, SWIG example; manages a FIFO queue of characters
2
3  char *fifo; // the queue
4
5  int nfifo = 0, // current length of the queue
6      maxfifo; // max length of the queue
7
8  int fifoinit(int spfifo) // allocate space for a max of spfifo elements
9  { fifo = malloc(spfifo);
10    if (fifo == 0) return 0; // failure
11    else {
12       maxfifo = spfifo;
13       return 1; // success
14    }
15 }
16
17 int fifoput(char c) // append c to queue
18 { if (nfifo < maxfifo) {
19       fifo[nfifo] = c;
20       return 1; // success
21    }
22    else return 0; // failure
23 }
24
25 char fifoget() // delete head of queue and return
26 { char c;
27    if (nfifo > 0) {
28       c = fifo[0];
29       memmove(fifo,fifo+1,--nfifo);
```

```
30         return c;
31      }
32      else return 0; // assume no null characters ever in queue
33   }
```

除了*.c*文件外，SWIG还需要一个接口文件，在本例中是*fifo.i*。该文件由全局符号组成，用SWIG样式列出一次，用C样式列出一次。

```
%module fifo

%{extern char *fifo;
extern int nfifo,
           maxfifo;
extern int fifoinit(int);
extern int fifoput(char);
extern char fifoget(); %}

extern char *fifo;
extern int nfifo,
           maxfifo;
extern int fifoinit(int);
extern int fifoput(char);
extern char fifoget();
```

为了编译该代码，先运行`swig`，它产生一个额外的*.c*文件和一个Python文件。然后使用GCC和`ld`产生一个*.so*共享的目标动态库。下面是Make文件。

```
_fifo.so:  fifo.o fifo_wrap.o
        gcc -shared fifo.o fifo_wrap.o -o _fifo.so

fifo.o fifo_wrap.o:  fifo.c fifo_wrap.c
        gcc -fPIC -g -c fifo.c fifo_wrap.c -I/usr/include/python2.4

fifo.py fifo_wrap.c:  fifo.i
        swig -python fifo.i
```

这个库被作为一个模块导入Python，如下面的测试程序中所示。

```
# testfifo.py

import fifo

def main():
   fifo.fifoinit(100)
   fifo.fifoput('x')
   fifo.fifoput('c')
   c = fifo.fifoget()
```

```
    print c
    c = fifo.fifoget()
    print c

if __name__ == '__main__': main()
```

该程序的输出应为“x”和“c”，但是我们发现输出为空：

```
$ python testfifo.py




$
```

为了使用GDB，记住你在运行的实际程序是Python解释器python。因此在该解释器上运行GDB。

```
$ gdb python
```

现在，我们将在附加的FIFO库中设置一个断点，但是这个库还没有加载。直到Python解释器执行了如下代码才会发生加载：

```
import fifo
```

幸而，我们可以要求GDB在库中的某个函数上停止：

```
(gdb) b fifoput
Function "fifoput" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (fifoput) pending.
```

现在运行解释器，其参数为测试程序*testfifo.py*。

```
(gdb) r testfifo.py
Starting program: /usr/bin/python testfifo.py
Reading symbols from shared object read from target memory...(no debugging
symbols found)...done.
Loaded system supplied DSO at 0x164000
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
[Thread debugging using libthread_db enabled]
[New Thread -1208383808 (LWP 15912)]
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
```

```
Breakpoint 2 at 0x3b25f8: file fifo.c, line 18.
Pending breakpoint "fifoput" resolved
[Switching to Thread -1208383808 (LWP 15912)]

Breakpoint 2, fifoput (c=120 'x') at fifo.c:18
18      {  if (nfifo < maxfifo)  {
```

你现在可以做已经非常熟悉的事：

```
(gdb) n
19              fifo[nfifo] = c;
(gdb) p nfifo
$1 = 0
(gdb) c
Continuing.

Breakpoint 2, fifoput (c=99 'c') at fifo.c:18
18      {  if (nfifo < maxfifo)  {
(gdb) n
19              fifo[nfifo] = c;
(gdb) p nfifo
$2 = 0
```

问题就在这里。每次试图向队列中添加字符时，只要重写前面添加的字符。第19行应为：

```
fifo[nfifo++] = c;
```

一旦进行了修改，代码就能正确工作。

8.5　汇编语言

GDB和DDD在调试汇编语言代码时极其有用。本节将描述要记住的一些特殊情况。
以文件*testff.s*中的代码为例。

```
1   # the subroutine findfirst(v,w,b) finds the first instance of a value v
2   # in a block of w consecutive words of memory beginning at b, returning
3   # either the index of the word where v was found (0, 1, 2, ...) or -1 if
4   # v was not found; beginning with _start, we have a short test of the
5   # subroutine
6
7   .data  # data segment
8   x:
9         .long   1
10        .long   5
11        .long   3
12        .long   168
```

```
13        .long   8888
14   .text  # code segment
15   .globl _start  # required
16   _start:  # required to use this label unless special action taken
17        # push the arguments on the stack, then make the call
18        push $x+4  # start search at the 5
19        push $168  # search for 168 (deliberately out of order)
20        push $4  # search 4 words
21        call findfirst
22   done:
23        movl %edi, %edi  # dummy instruction for breakpoint
24   findfirst:
25        # finds first instance of a specified value in a block of words
26        # EBX will contain the value to be searched for
27        # ECX will contain the number of words to be searched
28        # EAX will point to the current word to search
29        # return value (EAX) will be index of the word found (-1 if not found)
30        # fetch the arguments from the stack
31        movl 4(%esp), %ebx
32        movl 8(%esp), %ecx
33        movl 12(%esp), %eax
34        movl %eax, %edx # save block start location
35        # top of loop; compare the current word to the search value
36   top:  cmpl (%eax), %ebx
37        jz found
38        decl %ecx  # decrement counter of number of words left to search
39        jz notthere  # if counter has reached 0, the search value isn't there
40        addl $4, %eax  # otherwise, go on to the next word
41        jmp top
42   found:
43        subl %edx, %eax  # get offset from start of block
44        shrl $2, %eax  # divide by 4, to convert from byte offset to index
45        ret
46   notthere:
47        movl $-1, %eax
48        ret
```

这是Linux Intel汇编语言，使用的是AT&T语法，但是熟悉其他Intel语法的用户应当发现该代码容易使用。（GDB命令`set disassembly-flavor intel`会导致GDB以Intel语法显示其`disassemble`命令的所有输出，这类似于NASM编译器所使用的语法。顺便提及，由于这是Linux平台，因此程序在Intel CPU的32位平面模式下运行。）

正如注释所指示的，子例程`findfirst`从内存中一个指定的连续单词块中发现指定值第一次出现处。该子例程的返回值是找到该值的单词索引（`0,1,2,`…），如果没有找到，则返回`-1`。

该子例程预期参数被放在栈上，因此栈在输入时类似如下形式：

`address of the start of the block to be searched`	要搜索块开始处的地址
`number of words in the block`	块中的单词数量
`value to be searched`	要找的值
`return address`	返回地址

说明 Intel栈向下延伸，即向内存中的地址0增长。地址较小的单词出现在图中较下方。

为了引入可以使用GDB查找的一个程序错误，我们故意搅乱“主”程序调用序列中的元素。

```
push $x+4  # start search at the 5
push $168  # search for 168 (deliberately out of order)
push $4  # search 4 words
```

调用之前的指令则应当是：

```
push $x+4  # start search at the 5
push $4  # search 4 words
push $168  # search for 168
```

正如在编译C/C++代码时将`-g`选项用于GDB/DDD一样，这里在汇编层上使用`-gstabs`。

```
$ as -a --gstabs -o testff.o testff.s
```

这时产生了一个目标文件*testff.o*，并且并排输出了汇编源代码以及对应的机器码。还显示了数据项的偏移量及对于调试过程可能有用的其他信息。

然后我们链接：

```
$ ld testff.o
```

这时产生了一个可执行文件，默认名为*a.out*。

让我们在GDB下运行该代码。

```
(gdb) b done
Breakpoint 1 at 0x8048085: file testff.s, line 18.
(gdb) r
Starting program: /debug/a.out
Breakpoint 1, done () at testff.s:18
18              movl %edi, %edi  # dummy for breakpoint
Current language:  auto; currently asm
(gdb) p $eax
$1 = -1
```

我们看到，可以通过美元符号前缀引用寄存器。在本例中`$eax`表示EAX寄存器。但是，该寄存器中的值为`-1`，表示在指定块中没有找到要找的值`168`。

当调试汇编语言程序时，要做的第一件事情是检查栈的精确性。因此，让我们在该子例程上

设置一个断点，然后当到达那里时检查栈。

```
(gdb) b findfirst
Breakpoint 2 at 0x8048087: file testff.s, line 25.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /debug/a.out
Breakpoint 2, findfirst () at testff.s:25
25              movl 4(%esp), %ebx
(gdb) x/4w $esp
0xbfffd9a0:    0x08048085    0x00000004    0x000000a8    0x080490b4
```

这个栈当然是内存的一部分。因此要检查它，必须使用GDB中检查内存的x命令。这里，我们要求GDB显示从栈指针ESP指示的位置开始的4个单词（注意上面的栈图显示了4个单词）。x命令会显示内存，以便增加地址。这正是所需要的，因为在Intel架构上和在很多其他架构上一样，栈是向0增长的。

从上面显示的栈图中可以看到第一个单词应为返回地址。有多种方式可以检查这一预期。一种方法是使用GDB的disassemble命令，该命令列出了汇编语言指令（机器码的反向翻译）及其地址。单步进入子例程，然后可以检查栈上第一个单词的内容是否与调用后函数的地址相匹配。

检查后将发现是匹配的。然而，第二个数字4本应为被搜索的值（168），而实际上是搜索块的大小（4）。从这一信息可以很快意识到我们不小心在调用前交换了这两个push指令。